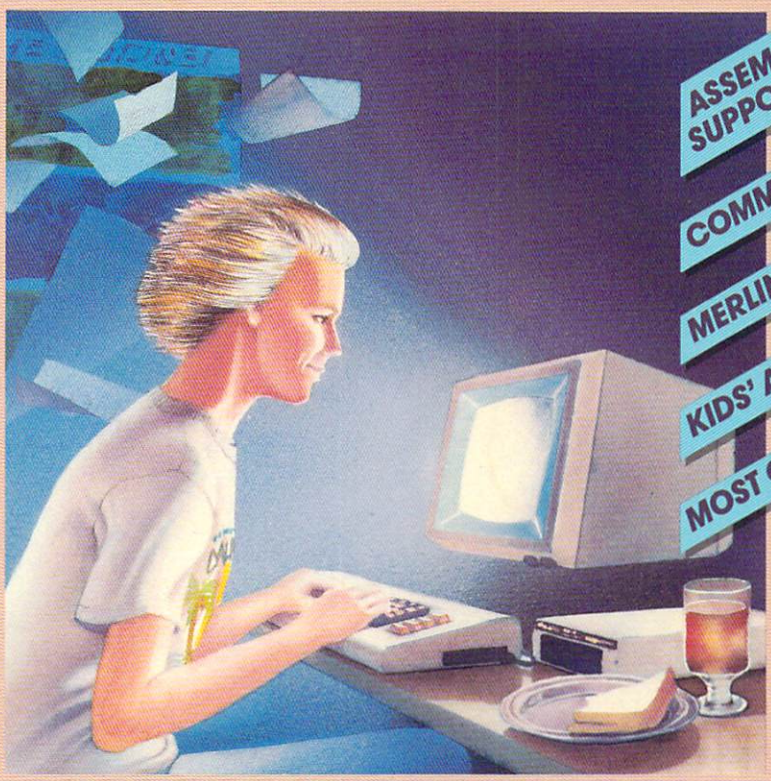
$14.95

ASSEMBLY LANGUAGE

FOR KIDS

COMMODORE 64



ASSEMBLERS SUPPORTED

COMMODORE

MERLIN 64

KIDS' ASSEMBLER

MOST OTHERS

by

WILLIAM B. SANDERS

FREE KIDS' ASSEMBLER INCLUDED

ASSEMBLY LANGUAGE

FOR KIDS:

COMMODORE 64

by William B. Sanders, Ph.D.
San Diego State University

Microcomscribe
Literate Microcomputer Documentation
8982 Stimson Court
San Diego, California 92129
619/484-3884 or 619/578-4588

Library of Congress Cataloging in Publication Data
Sanders, William B.
Assembly Language For Kids: Commodore 64
Includes index
1. Commodore 64 Computer
2. Assembly Language (Computer program language)
I. Sanders, William B., 1944-
II. Title: Assembly Language for Kids:
Commodore 64

ISBN 0-931145-00-7

San Diego, California

Manufactured in the United States of America

Cover design by Dyna Pac

Typesetting by GD Enterprises

ASSEMBLY LANGUAGE FOR KIDS : COMMODORE 64

TABLE OF CONTENTS

PREFACE VI

CHAPTER 1: INTRODUCTION 1

What This Book is About 1
Who This Book is For 2
Why Use Assembly Language 3
What is An Assembler? 5
Machine and Assembly Language 6
Assemblers Covered in this Book 8
The Commodore 64 Macro Assembler Development System 8
Merlin Assembler 8
The Kids' Assembler (Included in the book) 9
Other Assemblers not Covered 9
BASIC and Assembly Language 10
Machine Subroutines from BASIC 13
Setting Up 17

CHAPTER 2: USING AN ASSEMBLER 21

ASSEMBLERS IN GENERAL 21
The Standard Parts of An Assembler 23

Standard Editor/Assembler Format 27
USING THE KIDS' ASSEMBLER 29
Creating and Saving Programs 43
Special Conventions in Opcodes 47
Loading and Executing Programs 48
Some Examples 50

CHAPTER 3: THE MERLIN 64 ASSEMBLER 53
Using the Editor/Assembler 53
Loading and Running Programs 67
THE SOURCEROR 68
MERLIN'S MONITOR 70
Some Examples 70

CHAPTER 4: THE COMMODORE 64 MACRO ASSEMBLER DEVELOPMENT SYSTEM 73
THE PARTS 73
EDITOR64 74
Added Editing Functions on EDITOR64 79
ASSEMBLER64 80
LOADERS 82
THE MONITORS 83
Some Examples 88

CHAPTER 5: STRANGE NEW NUMBERS 91
Decimal, Binary and Hexadecimal Numbers 91
Going Between Number Systems 96
Using Conversion Charts 96
Conversion Programs 98
How Not To Worry About Numbers 101

CHAPTER 6: WHAT'S IN YOUR MICROPROCESSOR? . 103
What's a Register? 103
The Accumulator 104
The X and Y registers 105
The Processor Status Register (Status Register) 105
Negative Flag 106
oVerflow Flag 107
Break Flag 107
Decimal Flag 107

Interrupt Flag 107
Zero Flag 107
Carry Flag 108
Stack Pointer 108
Program Counter 110
Input/Output Port 110

CHAPTER 7: MEMORY AND STORAGE 113
Line numbers and Addresses : A Comparison 113
ROM and RAM memory 116
MINI-MONITOR 119
Backward Numbers : Low-Byte / High-Byte Storage 121
Bytes, Opcodes and Addressing Modes 123
SUMMARY 124

CHAPTER 8: JUMPING IN 125
Where to Stick Your Programs 125
Auto-Placement With Kids' Assembler 125
ORG Pseudo-Opcode In Merlin 125
Commodore Assembler's * = function 126
Visiting Built-in Subroutines with JSR 126
Getting Out with RTS 128
Loading up with LDA 129
Implied, Immediate and Absolute Addressing Modes ... 132
Storing with STA 134
Storage in Empty, Unused RAM 134
Storage in "Soft-Switch" Addresses 135
Storage on Your Screen 140
SUMMARY 142

CHAPTER 9: USING THE X AND Y REGISTERS 143
How to use the X and Y Registers 143
Transfers With TAX, TAY, TXA and TYA 145
Incrementing and Decrementing with
INX, INY, DEX and DEY 148
From X and Y to Memory with STX and STY 150
Addressing Modes with the X and Y Registers 153
Indexed Absolute Addressing 154
Indexed Indirect Addressing 157
Indirect Indexed Addressing 161

CHAPTER 10: LOOPS AND BRANCHES 165
Program Structure 165
Sequential 165
Loops 166
Branches 166
How BASIC Logic Works in Assembly Language 167
Looping to Save Programming Time 168
Indexing With Loops 172
Nested Loops 175
Branching Forward With JMP, BEQ and BNE 177
SUMMARY 181

CHAPTER 11: ADDING AND SUBTRACTING 183
Incrementing and Decrementing Memory: INC and DEC .. 183
Adding and Subtracting in the In the Accumulator:
ADC and SBC 186
Using CLC and SEC 187
SUMMARY 191

CHAPTER 12: INTERACTING WITH ASSEMBLY
LANGUAGE PROGRAMS 193
Introduction 193
Reading Input From the Keyboard 194
Joystick Control 203
The EOR instruction 205
Making Messages: ASC and .BYTE 221
Message Maker for Kids' Assembler 215
SUMMARY 218

CHAPTER 13: HOT GRAPHICS 218
Introduction 219
Low Resolution Graphics 220
Saving Plotted Lines 228
Animation 230
External Control of Movement 237
SUMMARY 246

CHAPTER 14: BLAZING SPRITES AND
MONSTROUS SOUNDS 247
Sprite Graphics 247
Sprite Creation 251

Sprite Building 260
Kids' Sprite Assembler 265
Full Horizontal Movement 272
Sprite Expansion 272
Assembly Sounds 274
SUMMARY 278

CHAPTER 15: DOWN THE ROAD 279
Introduction 279
Merging Subroutines 280
Appending With Merlin 280
Appending With Commodore EDITOR64 280
Appending and Inserting Subroutines 285
Getting to Know the Other Opcodes 289
Resources for Learning more About Assembly Language Programming on the Commodore 64 290
User Groups 290
Reference Books 291
How-To Books 292
Magazines 293
You're On Your Own 294

APPENDICES 297
APPENDIX A: KIDS' ASSEMBLER 299
APPENDIX B: 6510 OPCODES 309
APPENDIX C: MEMORY MAP 1: DIAGRAM 315
APPENDIX D: MEMORY MAP 1: PLACES TO VISIT 317
APPENDIX E: BASIC TOKEN CHART 321
APPENDIX F: HEXADECIMAL-DECIMAL CONVERSION CHART 323
APPENDIX G: DECIMAL-HEXADECIMAL CONVERSION CHART 325
APPENDIX H: SCREEN STORAGE ADDRESS TABLE 327
APPENDIX I: COLOR STORAGE LOCATION TABLE 329
APPENDIX J: ASCII CODE 331
APPENDIX K: SCREEN STORAGE DISPLAY CODES 333

INDEX 335

PREFACE

Learning assembly language programming is a challenge not everyone will accept. It is not as easy to learn as BASIC, and it requires learning how to work an assembler as well as learning assembly language itself. Many who attempted to learn assembly language have been stopped by lack of an assembler, lack of instructions on how to work an assembler or lack of instructions for learning assembly language on the Commodore 64. If any one problem didn't prevent learning the language, the combination of problems did.

Since these problems are real, I decided that it would be a good idea to create a book that provided solutions to all of these hurdles. First, the book would supply a simple assembler that would be instructional as well as functional. Secondly, it would explain how to work the most popular assemblers for the Commodore 64, including the one provided in the book. Finally, the book would take assembly language a step at a time, explaining how to program using assembly language instructions. That's what this book does.

As the title implies, this book was designed for kids. This doesn't mean the book is for children, but rather it's for kids who want to enjoy learning something. I would have never learned BASIC or assembly language if at some point I didn't start enjoying myself. The same principle applies here. If we try to have a good time while learning assembly language, it becomes an interesting challenge instead of boring work. There are lots of examples that show something about assembly language in a fun way.

ACKNOWLEDGEMENTS

If ever there were a group of people who were willing to help an author, it's my Commodore computer club, the San Diego Pet Users Group. Jane Campbell, Don Johnson and Barbara Prouty answered plenty of questions and provided lots of ideas. Fellow authors, Guy Grotke, who wrote *Intermediate Commodore 64,* and David Miller, author of *Commodore 64 Files,* were equally helpful with information and suggestions. Eric Goez is responsible for teaching me most of what I know about assembly language programming. Roger Wagner, author of *Assembly Lines: The Book,* served as an excellent editor. My own kids, Billy and David, were guinea pigs whenever I could detach them from MTV. My wife Eli, as usual, was a good sport about the whole thing. Naturally, any fault with this book lies with the author, not with those who so willingly helped.

CHAPTER 1

INTRODUCTION

What This Book Is About

This book is about creating machine language programs with an assembler. So the first thing we will learn how to do is to use an assembler. If you do not have an assembler, I've included one in this book you can use. You will learn how to write assembly language with specific assemblers. We will go through their use step by step covering only their basic features. Most assemblers have advanced features you can use, but we will not be going into these aspects of assemblers. Once you are comfortable on this level, you can read more advanced books and documentation for special features of the various assemblers. If you already have an assembler, and you know how to use it, you can skip Chapters 2-4 since their sole purpose is to explain how to operate different assemblers.

Secondly, this book will explain using assembly language coding procedures. There are many different operations you can perform with an assembler. In fact there are 151 different machine opcodes accessed by 56 assembly language opcodes. An opcode is an instruction something like BASIC statements. However, we are not going to cover all the assembly language opcodes. Some opcodes involve complicated operations, and we're not going to get overly advanced

in this book. It is more important to learn how to use the fundamental operations well and understand their use clearly than try and learn everything at once and not understand what you're doing. Nevertheless, when you are finished with this book, you will be able to write assembly language programs. Furthermore, we are only going to deal with the Commodore 64. If you have a different computer, even a VIC 2Ø, you should get a different book.

Finally, we're going to have to spend some time on how your microprocessor handles code, different number systems and storage. To be honest, this isn't a lot of fun, but once we're through, you'll understand a lot more about your computer and how to handle machine and assembly language operations. This is all covered in Chapters 5-7. If you know about binary, hexadecimal and decimal number conversions and how the whole thing works in your microprocessor, skip these chapters. (There's no sense in spending time on something you already understand.) From Chapter 8 on, we start writing assembly language programs.

Who This Book is For

This book is for kids who want to begin programming in assembly language. You might want to know the difference between a general book for beginners and one for kids. In a lot of ways there is no difference, especially if this is your first venture into assembly language. However, we're not going to get too serious, and the emphasis will be on having a good time learning assembly language programming instead of a comprehensive guide to professional programming. Also, we're going to play a lot with little routines that do crazy things to your Commodore 64 instead of writing programs for the sequential development of applications. This means we're not going to be getting into abstract structures and theories about microprocessors. Instead the emphasis will be on learning by doing. Most of the routines will display something to the screen so that you can see what's going on. This means we will be doing a lot with text and graphic displays and not so much with mathematical manipulations important for business applications. Besides, its more entertaining to watch a sprite go screaming across the screen than adding up a column of numbers.

As a warning to those of you who just got your Commodore 64, you should know that assembly language programming is a lot more difficult to master than BASIC. In fact, if you do not know how to program in BASIC, I would strongly recommend you learn it before tackling machine and assembly language. Many of the examples and explanations are based on the assumption that you have a working knowledge of BASIC. There are several good books for kids learning BASIC, including KIDS TO KIDS ON THE COMMODORE 64 and KIDS AND THE COMMODORE 64, and you should read them before going on here. However, if you know how to program in BASIC, you're halfway to understanding machine language since you can transfer much of the BASIC programming logic to assembly language programming.

Finally, since this book is designed for kids, you might expect to be talked to as adults tend to talk to kids. The last thing I want to do is to talk down to anyone. We'll go slowly and clearly, and we'll have some fun, but don't expect to be treated like a half-wit. Also, this book is not going to have a lot of tests to see if you got everything right. Books with little tests at the end of the chapters are fine if you like taking tests. You have enough tests in school; so you won't get any in this book. Instead we'll have lots of example programs that explain and illustrate the use of various assembly language programming techniques. By using your own imagination and playing around with the various opcodes, you can learn more than by taking tests.

Why Use Assembly Language?

Most books on assembly and machine language explain the speed and compactness of machine code as the major reason to use it instead of BASIC. That's certainly true, but I like assembly language because you can really grab control of your Commodore 64, impress your friends and make a fortune. If you have ever played a good arcade game, chances are it was written in machine language with an assembler. The sound and speed of movement just aren't possible with BASIC. If you ever shelled out money for such a game, who do you think is getting it? A good chunk of the money goes to the person who wrote the game program. Believe it not, I

know kids who can retire when they graduate from high school with the money they've made programming with machine language. Several other kids I know have part-time jobs after school working for businesses requiring customized programs written with assemblers. They may not make a fortune, but they get paid better than the kids working in fast food joints and enjoy their work a heck of a lot more. Now don't expect to write a best-selling arcade game after you read this book or even get a paying job as a programmer, but you will get started, and even the best programmers had to start somewhere.

Now if you're not interested in making money as a professional programmer, you can treat assembly language programming as an adventure game. The whole purpose of adventure games is to find a treasure hidden in a maze of caverns, tunnels, castles and dungeons. Believe me, once you enter the caverns of your Commodore 64's memory with its 64,ØØØ caves of RAM plus all the tunnels of ROM, you'll need all your wits to save your neck. What's more, once you understand the memory, you will have all kinds of control even the best adventure games cannot provide.

Finally, if you have any friends who know even a little assembly language programming, you got to admit that you're impressed. Kids who can handle machine code have a certain magic about them that draws admiration. I admit it; it's fun to impress people and show off. You don't need any brass band or be a loud mouth with assembly language. Just crank up some code and let her rip.

Assembly language programmers have style!

This book is not meant to disparage BASIC; I use BASIC all the time. It's quicker to write a program in BASIC and less difficult to de-bug. As we will see later in this chapter, one of the most useful applications of assembly language programs is to write subroutines for BASIC. So instead of abandoning BASIC altogether, you can use assembly language programs to enhance your BASIC programs.

What is an Assembler?

An assembler is a program that allows you to enter machine language using a standard set of "mnemonics." What?!! First we better explain what "mnemonics" are. Pronounced 'ni-mon-iks', mnemonics are aids to help remember something. All of the instructions for your 651Ø microprocessor are given with three letter mnemonics. For example, one commonly used instruction is 'LDA.' The mnemonic LDA stands for LoaD Accumulator. The machine language opcode for LDA in the immediate mode is the hexadecimal number $A9 or decimal 169. Since it's a lot easier to remember LDA instead of $A9 or 169 when you want to LoaD Accumulator, programmers prefer using assemblers instead of entering machine code directly. (In fact, a lot of machine code is entered with POKEs from BASIC. A POKE X,169, where 'X' is some address in memory, could very well be a machine language instruction from BASIC to perform an LDA.)

Secondly, assemblers keep track of everything for you. When machine language is entered into your machine, you must use sequential addresses for your instructions and values. Let's suppose you want to enter your machine language program in addresses 49152 to 49162 ($CØØØ - $CØØA). Depending on what instructions, modes, values and addresses are used, more or less memory will be taken up. Instead of having to figure out all of these elements as you go along, the assembler does it for you. For example, if you use the LDA instruction in the immediate mode, you take up two addresses. However, if you use LDA in the absolute mode, you need three addresses. Furthermore, while LDA in the immediate mode has a machine language value of $A9 (169 decimal), LDA in the absolute mode has a value of $AD (173 decimal.) This is all understood by your assembler, and you don't have to do anything other than give it

the mnemonic instruction and mode to have it do what its supposed to do.

If you do not understand everything just discussed, DON'T WORRY! Essentially, all that I'm trying to tell you is that an assembler makes programming in machine language a lot easier! You're not expected to grasp everything all at once, and what is unclear at this point will make a lot more sense as you start working with an assembler. The trick is to forge ahead and start working with an assembler and the programs and instructions in this book. You will remember when you started learning BASIC that everything was not clear until you started experimenting, learning special tricks and writing your own programs. The same is true with assembly language. Practice will lead to understanding.

Machine and Assembly Language

By now, you probably realize that assembly and machine language are two sides of the same coin. Assembly language uses mnemonic codes to enter numeric machine code. In addition, an assembler keeps track of everything needed for a machine program to execute. For example, let's look at three programs that all do the same thing. First, we'll look at how the program would be entered with an assembler, then we'll see what it looks in machine language, and then we'll see how it would be entered from BASIC directly into machine code. Each method ends up with the same result, and I'll let you decide what method is the most understandable and least difficult. All the program does is to jump to a subroutine located inside your computer at hexadecimal address $E544 (58692 decimal) and then returns to BASIC. The subroutine clears your screen and puts the cursor into the upper left hand corner.

Assembly Language

```
ORG $C000 ; Begin storing program here
JSR $E544 ; Jump to subroutine at $E544
RTS ; Return to BASIC
```

Machine Language

```
$C000 $20
$C001 $44
$C002 $E5
$C003 $60
```

BASIC

```
10 C = 58692 : REM DECIMAL VALUE FOR $E544
20 LB = C – INT(C/256) * 256 : REM LO-BYTE
30 HB = INT(C/256) : REM HI-BYTE
40 POKE 49152,32 : REM JSR
50 POKE 49153,LB
60 POKE 49154,HB
70 POKE 49155,96 : REM RTS
```

If you think the assembly language program is the simplest you're right! Even with the REM statements, it's difficult to know what's going on in the BASIC program, and at this stage of the game, the machine listing should make no sense at all! In the assembly program, all you have to do is to tell it where to start placing the code with the ORG directive (or some similar instruction, depending on the assembler), and then using the mnemonic codes, tell it what you want it to do. Since the program is very small, using only four addresses or bytes, you can imagine how difficult it would be to enter everything in machine language or POKE it in with BASIC with longer programs. In magazine listings, you've probably seen BASIC listings with POKEs and then a mile of DATA statements with the values to be POKEd in. The programmer probably had to first write the program using an assembler and then "disassemble" it with PEEKs to get all those values.

Just for fun, key in the BASIC program and RUN it. When you're finished enter:

```
SYS 49152
```

Your screen will clear. Of course it's a lot easier to PRINT "{CLR/HOME}" from BASIC to do the same thing, but a program line in BASIC with the instruction to clear the screen and home

the cursor takes up a lot more room in your computer's memory. Later in this chapter we'll compare the amount of memory used to see precisely how much more space is used by BASIC and why machine code is so much faster.

ASSEMBLERS COVERED IN THIS BOOK

Since learning how to use a specific assembler requires special instructions, I decided to go into detail on the operations of certain ones that you may already have, are recommended or, if you don't have one, the one in this book. If you have an assembler and know how to use it, you can skip this section and Chapters 2-4. If, on the other hand, you bought an assembler not covered in this book but cannot figure out how to work it, take a look at Chapters 2-4, and your's may work in a similar manner to ones we discuss. If all else fails, you can always key in the Kids' Assembler in the book and use it.

The Commodore 64 Macro Assembler Development System

Chances are if you own an assembler already, you probably have the one made by Commodore. I found it to be difficult to understand, but once I got used to it, it works fine. You can edit, assemble and load your code. It also comes with a machine language monitor for entering assembly or machine language routines. However, it does take a lot of work to load all the different files necessary to do the different tasks required in writing assembly programs. In Chapter 4, we'll go through its use step by step so that you can use it effectively.

The Merlin 64 Assembler

This assembler is my own personal choice, and if you don't have an assembler and want to get a good quality one inexpensively, I would suggest this one. Roger Wagner Publishing has made a special deal for kids. In the back of this book, you will find a coupon for the Merlin 64 Assembler at a reduced cost. You can also find it in your local stores.

The Kids' Assembler

In Chapter 2 and Appendix A there are BASIC listings for an assembler written especially for kids. (In Chapter 2, we go through the program in parts to see how it assembles code, and in the Appendix, the listing contains all of the instructions used by the 651Ø microprocessor. For the time being, use the one in Chapter 2 since it deals with just those instructions we will be using.) Since I wrote this assembler, I can criticize it. It has a minimal editor, it uses non-standard opcodes, and because it is written in BASIC it is slow. However, it does have a certain amount of error trapping so that if you enter an illegal value or instruction, it will let you know right away. It also shows you where your code is going in decimal values. Furthermore, since you will key it in yourself, it will help you understand what's going on. Best of all, it's very simple to use, and it's free! Sooner or later, though, you will want to get a good assembler with an editor and monitor.

Other Assemblers Not Covered in this Book

There are several magazines that provide listings of machine language and BASIC assemblers. Magazines such as COMPUTE!'s GAZETTE and THE COMMANDER have had listings of assemblers, editors and monitors for the Commodore 64. Each assembler, like the ones discussed in this book, have their own unique characteristics and manner of handling assembler instructions. Likewise, several companies make commercial assembler/editors, but there are far too many to explain each one's use here. Since the documentation for these assemblers ranges from the fairly clear to the almost incomprehensible, you might have a problem understanding how to use them. Therefore, I have provided a short section at the beginning of Chapter 2 entitled "Assemblers in General" to give you a running start on their use. Actually, we will be dealing with entering code with "editors" in that section, but most assemblers have built-in editors so that when you run the assembler, you also have access to the editor. (The Commodore Editor64 and Assembler64 are separate co-resident files, however.) After reading that section, re-read the documentation with your assembler/editor and you can probably get everything going.

BASIC and Assembly Language

As we have seen, there are some connections between BASIC and assembly language. In fact, after going through a number of interpretive steps, the programs you write in BASIC are ultimately executed in machine language. Since there are so many interpretations, though, it takes longer to execute a BASIC program than one written in assembly language. Let's take a look at our little machine language program to clear the screen and one written in BASIC that does the same thing.

```
JSR $E544
RTS
```

```
10 PRINT CHR$(147) ◄Leave this in memory
```

We used only four addresses for our machine language program.

Address		Opcode or Operand
Hex	Dec	
$C000	49152	$20 ◄Machine code for JSR
$C001	49153	$44 ◄Low byte of jump address
$C002	49154	$E5 ◄High byte of jump address
$C004	49155	$60 ◄Machine code for RTS

Now, how do we examine the memory used by our BASIC program? The most simple way is to subtract the amount of free memory we have from the 38911 bytes that are free when we start up the Commodore 64. You remember the FRE(Ø) statement in BASIC. If you PRINT FRE(Ø) you will be presented with the amount of available memory. Since this is a negative number, we have to add it to 65536 to see how much memory is available. Then by subtracting that amount from 38911 we will find how much our one line program used. Therefore enter:

```
PRINT 38911 - (65536 + FRE(Ø)) {RETURN}
```

You should have gotten '17', the number of bytes used by the program. That's over four times the amount used by our assembly language program!

To actually see the code in your computer in a BASIC program, we will "disassemble" the BASIC code. In your Commodore 64, BASIC programs begin at location $800 (2048 decimal). To find the end of the BASIC program and the start of variables, we look at a 'pointer' at locations $2D-$2E (45-46 decimal). A pointer is pretty much what it implies - it points to an address. By PEEKing at the locations between 2048 and the address stored in the pointer to the end of the program, we should be able to see the entire BASIC program. So first, we have to find the end of the program. We do that by PEEKing the value stored in locations 45-46 using the following algorithm:

```
PRINT PEEK(45) + PEEK(46) * 256 {RETURN}
```

= = EVERYTHING IS BACKWARDS! = =

Now if you're as smart as you look, you're probably wondering why the heck we used that weird second PEEK. Why did we have to multiply it by 256 instead of just a regular PEEK? If you look at the machine code listing, you will see that when we jumped to the subroutine at $E544, the code was entered as $44 $E5 in two consecutive addresses. That's called low-byte, high-byte storage. Therefore, when we pull a number out of two consecutive storage address, we convert it to the correct decimal value by multiplying the high byte (the second address) by 256. Then by adding the high and low bytes, we get the correct decimal number. You don't have to worry about understanding all of this now, but just remember that numbers are stored backwards! In Chapter 3, we'll explain more about this feature of your Commodore 64's memory.

You should have gotten '2064'. After all, we know that the BASIC program used 17 bytes of memory and BASIC programs begin at 2048. Thus, 2048 = 17 - 1 = 2064. (Since the program uses addresses 2048 to 2064 inclusive, we subtract 1 from used memory to get 2064.) Now, by PEEKing locations from 2048 to 2064 we can see our entire program. Enter the following:

```
FOR X = 2048 TO 2064 : PRINT X;PEEK(X) : NEXT X
{RETURN}
```

You get the following "disassembly":

Byte#	Address
1	2048 0 ◄Beginning of BASIC
2	2049 14
3	2050 8
4	2051 10 ◄Line number
5	2052 0
6	2053 153 ◄PRINT statement
7	2054 32 ◄Space
8	2055 199 ◄CHR$ function
9	2056 40 ◄Left parenthesis
10	2057 49
11	2058 52 ◄Mystery numbers
12	2059 55
13	2060 41 ◄Right parenthesis
14	2061 0
15	2062 0
16	2063 0
17	2064 88 ◄Beginning of variables

Just look at all of that to clear the screen! When you key in a BASIC program, you automatically key in a machine language program as well. While it is less work for you to key in BASIC, it's a lot more work for your computer as you can see.

When you key in a BASIC word, it is 'tokenized' into a machine code value. For example, we can see the token for PRINT to be 153. Likewise CHR$ is 199. The line number is there along with the left and right parenthesis, the space and some "mystery numbers" that are special codes for CHR$ values. (Appendix D has a complete listing of the BASIC tokens.) In the next section we'll do some tricks with these values that will give you power over BASIC that you never realized you had.

At this point you should be convinced that machine code is more compact than BASIC as far as your computer is concerned. Fur-

thermore, you should be able to deduct that if there is less code for the computer to read, it can execute it faster. Finally, you should see that when you write BASIC you are also writing a machine language routine. Now that you know all of that, let's see how you can use little machine code routines in your BASIC programs.

Machine Subroutines From BASIC

An old saying among programmers is that 10% of a program takes 90% of the time. For example, if you have a sort routine in your program, most of the time spent by your computer is sorting the numbers or strings you enter. The actual sorting routine may only take a few lines, but it is the routine that takes up the majority of your execution time. Since BASIC code can be entered very quickly compared with assembly code, a lot of programmers write machine language subroutines that can be called from a BASIC program using the SYS statement. In this way they can insert time-consuming routines, such as sorts, in machine code and speed up the slow part of the program without having to write the whole program in assembly language. What's more, you can put the machine code in one part of memory that will not interfere with the memory being used by BASIC. To see how this works, let's write a little machine code routine to change the color of your screen. We'll stick that in memory out of the way of BASIC and then call it from BASIC with SYS.

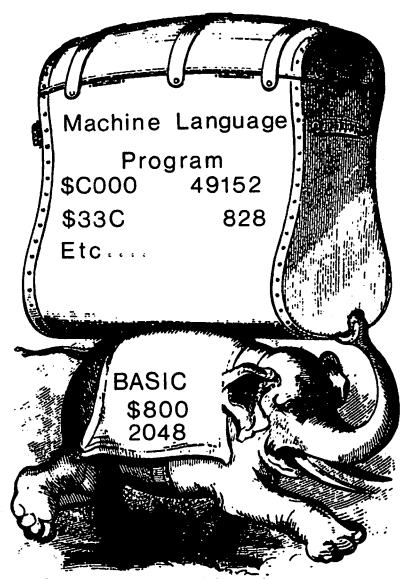
You probably know that decimal address 53281 holds the code for your background color. By POKEing 53281 with values between 0-15 you can change its color. In assembly language to do that, we would store a value in that location.

```
LDA #0
STA 53281
RTS
```

The above routine loads the accumulator with the number 0, the code for a black background, and sticks it in 53281. We'll put that routine at addresses 49152-49157 with the following BASIC program.

```
10 BC = 53281 : REM ADDRESS OF BACKGROUND
                  COLOR
20 LB = BC-INT(BC/256) * 256 : REM LOW BYTE
30 HB = INT(BC/256) : REM HIGH BYTE
40 POKE 49152,169 : REM LDA
50 POKE 49153,0 : REM BLACK COLOR CODE
60 POKE 49154,141 : REM STA
70 POKE 49155,LB : REM LOW BYTE ADRS
80 POKE 49156,HB : REM HIGH BYTE ADRS
90 POKE 49157,96 : REM RTS
```

Now RUN the program and then enter NEW. The BASIC program is no longer in memory because you entered NEW. Just to be sure you got rid of it, enter LIST {RETURN}. (If you get a listing, better enter NEW {RETURN} again.) The NEW command got rid of the BASIC program beginning in memory location 2048. But since you put your machine code way up in 49152, your machine code should still be there. Now enter the following BASIC program that will use the machine language program.



BASIC and machine language can run together.

```
10 PRINT CHR$(147)
20 INPUT "HIT RETURN TO SEE THE SCREEN GO
BLACK";A$
30 SYS 49152
40 PRINT "NOW IT'S BLACK!"
```

By the way, in case you didn't notice it, the BASIC program to enter machine code was pretty long. With an assembler, you'd just have the three lines to enter. So while BASIC in general is faster to program, it takes a lot more to write assembly language with a BASIC program than it does with an assembler.

Some Tricks to Get Started

To get off on the right foot, you should use your knowledge of machine language to do something impressive. In order to see a part of your new power, try the following.

```
10 GOTO 30
65535 PROGRAM WRITTEN BY ME
30 PRINT "TA DA!" : END
```

You can't do it. Right? As soon as you enter 65535 as a line number your Commodore 64 burps out a SYNTAX ERROR. Now try it again.

```
10 GOTO 30
20 PROGRAM WRITTEN BY ME
30 PRINT "TA DA!" : END
```

Normally, you'd get a SYNTAX ERROR in Line 20 since it is in an illegal format. However, Line 10 jumps to Line 30 so there's no problem. Now, let's PEEK at locations 43-44 to find the beginning of the program, and list a screen of locations and their values.

```
PRINT PEEK(43) + PEEK(44) * 256 {RETURN}
```

We get 2049. (Actually BASIC programs begin at 2048, but the interesting stuff starts at 2049.) Now let's "disassemble" the first 20 addresses of the BASIC program in memory.

```
FOR X = 2049 TO 2069 : PRINT X;PEEK(X) : NEXT
{RETURN}
```

You'll see the following (assuming you entered the program exactly as listed - even including the spaces).

```
2049 10
2050 8
2051 10 ◄Line number low byte
2052 0 ◄Line number high byte
2053 137 ◄GOTO token
2054 32 ◄Space
2055 51
2056 48
2057 0
2058 36
2059 8
2060 20 ◄Line number low byte
2061 0 ◄Line number high byte
2062 80
2963 82
2964 79
etc...
```

Notice that address 2060 has the value of 20 for line number 20. Right after that is a zero (0). Notice also that following the line number at address 2051, address 2052 has a zero (0). The zeros represent the high byte value for any line number. Now suppose we put the maximum value in the low byte *and* high byte addresses for Line 20. (The maximum value for any single byte is 255 or hex value $FF.) Let's see what happens:

```
POKE 2060,255 : POKE 2061,255 {RETURN}
```

Go ahead and LIST your program. You now have the program with the "illegal" line number. It should now look as follows:

```
10 GOTO 30
65535 PROGRAM WRITTEN BY ME
30 PRINT "TA DA!" : END
```

It will still RUN, but no matter how much you LIST it, it will not put the line numbers in numeric order. Also, unless you re-POKE the addresses storing the line number 65535, you can't get rid of line 65535. Go ahead and try. If you want to customize your programs and include a line with your name on it that most BASIC program-

mers cannot delete, use that little trick. The following is another version using the same trick. However, since we know the addresses storing the second line number, we can include a line in the program that will automatically change the line numbers.

```
10 GOTO 30
20 BY {YOUR NAME}
30 POKE 2060,255 : POKE 2061,255
40 PRINT CHR$(147) : LIST
```

When you RUN the program you screen will clear and you will see,

```
10 GOTO 30
65535 BY {YOUR NAME}
30 POKE 2060,255 : POKE 2061,255
40 PRINT CHR$(147) : LIST
```

From the above, you should now be able to see that what is stored in RAM space addresses can be changed. However, you have to know something about how your BASIC or machine language program is stored in order to make changes. If you can make those changes, then you have taken the first step in machine language programming. That's because the great bulk of assembly and machine language programming is placing instructions and information into addresses and using addresses to keep track of your program's information. Again, you probably don't understand everything at this point, but by practicing the examples and reading the explanations, things will soon come together. Also, by changing the examples and experimenting on your own, you will begin to understand how your computer works in your own terms. (Just for fun, why don't you see if you can change the little machine program that turned the background color of the screen to black. See if you can change it to different colors using values between 1-15. They're all listed at the end of this chapter.)

SETTING UP

Before you start using your assembler, be sure to make a back-up copy of it on a separate diskette or tape. There are a million ways to blow a disk or tape with machine code; so if you don't want to lose

your assembler, make a back-up copy of it. The two commercial assemblers discussed in this book, Commodore Assembler and the Merlin assembler, can be backed up. Be sure to put a copy of the assembler on your work disk and keep the master disk in a safe place.

Likewise, with the Kids' Assembler be sure to make a back-up of the disk or tape you SAVE it on. If you don't feel like keying it in, there's a coupon in the back of this book that you can send in and get a copy on disk for $1Ø. Whatever you do, though, make back-ups of your assembler.

There are two versions of the Commodore 64 we'll classify as the "old" and "new". In the old version, it is possible to store a character in the screen memory area, and it will appear on your screen after the screen has been cleared. On the newer versions, you have to load both the color and character after a JSR $E544 (clear the screen and home cursor) to see the characters. The old version will work with either, but the new version requires the color. Therefore, all of our programs will be set up for the newer version, but will work on both the old and new versions.

If you have a monitor, the default colors on your Commodore are not too clear. Therefore, both the Kids' Assembler and Merlin present your screen with a white background and border and black letters for better viewing. Since the default color of characters is white after a JSR $E544, a white on white display is invisible. Therefore, we will have to take care to keep the character colors straight.

If you have a color TV or color monitor and prefer the default Commodore colors, or some other combination, you can change them by POKEing the following addresses with values, represented by the variable X, between Ø-15.

```
POKE 53281,X ◄Background Color
POKE 5328Ø,X ◄Border Color
X's value can be from Ø-15
```

The colors are:

Ø Black	4 Purple	8 Orange	12 Gray 1
1 White	5 Green	9 Brown	13 Light Green
2 Red	6 Blue	1Ø Light Red	14 Light Blue
3 Cyan	7 Yellow	11 Gray 1	15 Gray 3

This all may be familiar to you, but just in case you did not know about changing your screen colors, you do now.

Now, let's get into assembly language!

CHAPTER 2

USING AN ASSEMBLER

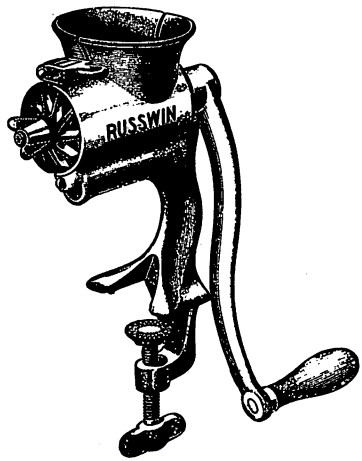
Assemblers in General

When I encountered my first assembler, the greatest problem I had was figuring out how to use it. There was no documentation for it since it was a public domain version I got from my club. Later on someone came along with the documentation, and after I read it, I still didn't know how to work the darned thing!

Finally I realized the problem with most assemblers and their documentation. THE DOCUMENTATION IS WRITTEN FOR PEOPLE WHO ALREADY KNOW ASSEMBLY LANGUAGE! For the kids (and adults) who do not know assembly language or other assemblers, it doesn't help to know that a whiz bang assembler has macro capabilities when you don't even know what a macro capability is. Therefore, to get started, we're going to demystify assemblers by exposing them for what they really are.

As we briefly mentioned in Chapter 1, all an assembler really does is to neatly and simply arrange machine language code. The code with which we work in assembly language can be broken down into three parts:

1. Addresses to store code.
2. Instructions to tell the computer what to do next. (Opcodes)
3. The exact location or mode relative to the instruction. (Operands)



Compiler

To understand a little about what's going on with an assembler, let's look at what assembled (or compiled) code is compared with the assembler instructions that made it get that way. (You remember our comparisons between machine and assembly code in Chapter 1.)

Compiled Machine Code

HEX		**DECIMAL**	
Address	**Code**	**Address**	**Code**
$C000	$20	49152	32
$C001	$44	49153	68
$C002	$E5	49154	229
$C003	$A9	49155	169
$C004	$A9	49155	8
$C005	$8D	49157	141
$C006	$21	49158	33
$C007	$D0	49159	208
$C008	$60	49160	96

Assembler Source Code

Line #	**Opcode**	**Operand**
1	JSR	$E544
2	LDA	#8
3	STA	$D021
4	RTS	

We already know that assembly language is less mysterious than machine code, but let me give you an idea of everything the assembler did to compile the code.

1. First, it looks at the opcode (the instruction) and the mode of the instruction and decides how many bytes it will take. For example, the instruction JSR (Jump to SubRoutine) takes up 3 bytes or addresses. The first byte is the machine opcode for JSR and the next two bytes are needed for the address ($E544). On the other hand, LDA in the immediate mode only needs two bytes and addresses; one for the operand and one for the value which has to be 255 ($FF) or less. The RTS opcode only takes a single byte or address.

2. Next, the assembler arranges the operands greater than 255 ($FF) and puts them in the lo-byte / high-byte configuration your computer uses.

3. Finally, it places these values all in memory in ascending order of addresses. ("Ascending" means from lower to higher addresses.) This is called the assembled or compiled code.

The work of an assembler should not be very clear to you right now, but later on the clouds will begin to clear and you'll say, "Oh yeah! Now I get it." In the meantime, just think of an assembler as a "code arranger" to make writing machine language simpler.

The Standard Parts of An Assembler

An assembler package sometimes has several different files representing the different things an "assembler" does. Actually, the "assembler" is only a single part of what most people call "assemblers." Generally, we deal with "editors" and "assemblers" as different parts of a single tool, but usually you never "see" the assembler do its work. Instead, you will be working primarily with the editor. Now let's take a look at all the parts:

EDITOR. The editor is used to enter what's called "source code." All of the mnemonic instructions, such as LDA and STA are accepted by the editor. When the source code is

assembled into machine code, it is called the "object code." In simple editors, such as the one in the Kids' Assembler, you can only enter source code. You can't edit it after its been entered. (It does have a good deal of error trapping so that if you enter illegal opcodes or operands, it will let you re-enter legal ones before continuing.) On more sophisticated editors, you can insert and edit lines, make global changes, move or copy big chunks of code and save and load source code. The Merlin Assembler and the Commodore 64 Assembler Development System have editors that can do some or all of those things plus more. When getting an "assembler", the real key to getting a good one or bad one lies in what you can do with the editor, not the assembler.

ASSEMBLER. Since the assembler part of an assembler is pretty much invisible to the user, and they all do essentially the same thing - compile code for you in machine language - there's not a whole lot to say about how good or bad the actual assembler is. However, since assemblers are so closely linked to the kind of editor one is using and how it handles code, invisible differences become apparent in the editor. A one-pass assembler takes the opcodes and operands and orders them into machine code either as soon as you enter the information in the editor (as does the Kids Assembler) or when you finish your program. A two-pass assembler is used in just about all commercial assemblers for the Commodore 64. First, the two-pass assembler finds the addresses and offsets for the labels, and then on the second pass, compiles it into machine code. On the Merlin Assembler and Commodore 64 Macro Assembler Development System, you can use labels in your editor; therefore, using two passes, the labels are automatically turned into the correct addresses and offsets. On the Kids' Assembler, since the editor will *not* accept labels, it compiles as soon as the operand is entered. What this means for the user is that it's a heck of a lot easier to use a two-pass assembler since you can do more things simply with the editor.

ASSORTED OTHER PARTS

1. **LOADERS** and **SAVERS.** Some assemblers have the loaders built into the editor while others have separate files for loaders. Most commercial assemblers will load the source code from the editor. Object code (the compiled code that runs) is saved either as SEQ or PRG files on your disk. (Tapes only accept object code as SEQ files or as DATA statements in PRG files.) If your object code is saved as a PRG file, you can load it from disk with LOAD "FILENAME",8,1 and then SYS its beginning address to make it run. If saved as SEQ files, it is necessary to have a driver program to first load the file into memory and then SYS it. A good assembly package will allow you to load and save source code and object code. It is especially important to have source code saved for developing larger programs with an assembler so that you can work on the program at different times. Likewise, it is far better to have your object code saved as a PRG file so that you can execute it without having to load a special loader program. The Kids' Assembler is good in that it saves object code as PRG files, but bad in that it saves the source code only as a SEQ file that cannot be reloaded into the editor. (That's why the programs in this book are relatively short!) The assembler by Commodore is good at saving source code that can be reloaded into the editor, but poor in that object code is saved as SEQ files requiring separate loader programs. The Merlin Assembler not only saves source code files, but it saves object code as PRG files. Furthermore, using the Sourceror program that comes on the Merlin Assembler disk, you can create source code from object code stored as PRG files. (That may not mean much to you now, but if you have a diskfull of object code saved with the Kids' Assembler, BELIEVE ME, you'll someday want the source code.)

2. **MONITORS.** Several assembler packages include monitors. (Monitors, in this instance, do not refer to the special TV sets for computers.) A monitor is a program that lets to examine, enter, change and execute machine code. In Chapter 1 we used a do-it-yourself BASIC monitor in examining the addresses

and code they contained. I didn't include a monitor in this book since I'm going to let you write your own! (There'll be plenty of tips on how to do it, though.) The Merlin and Commodore assembler packages contain very good monitors. The Merlin monitor is especially useful since it is co-resident with the editor/assembler.

3. **DISASSEMBLERS.** Disassemblers take assembled code in memory and lists it in rudimentary mnemonic and machine code. For example, a disassembly of our program in this chapter might looks as follows: (All numbers are in hexadecimal values.)

```
C000 20 44 E5     JSR $E544
C003 A9 08        LDA #$08
C005 8D 21 D0     STA $D021
C008 60           RTS $60
```

Disassemblers are very useful to see the relationship between machine code and assembly code. Take a look at the disassembly above with the listings near the beginning of the chapter to see if you can find the connection.

4. **OTHER GOODIES.** Assembler packages have all kinds of enhancements to help you with your assembly code. Some have de-buggers to help you find mistakes in your code. These vary from the built-in error trapping in the Kids' Assembler to ones that will let you find structural problems in your code. Secondly, source code generators are very helpful to examine how others have created object code, even ones created with different assembler packages. The Sourceror in the Merlin Assembler package is one such helpful program. Finally, many assembler packages come with a set of source code files of useful routines. These routines are not only valuable for seeing how a certain operation works, but they can be incorporated into your own programs to save you the time of developing them yourself.

STANDARD EDITOR/ASSEMBLER FORMAT

When you first start entering assembly language code, you'll do it from the editor. Just about all assemblers have four fields:

1. The label field.
2. The opcode field.
3. The operand field.
4. The comment field.

The fields arranged from left to right look like this in your editor:

LABEL OPCODE OPERAND COMMENT

As you enter the code in each field, to the left of the label field, line numbers appear. These are something like BASIC line numbers, but they usually have increments of 1 instead of 10 as in BASIC. (These line numbers are *not* compiled!) A typical editor entry would look liking the following:

```
LINE# LABEL   OPCODE  OPERAND  COMMENT
  1           LDX     #$0      ; Load X register
                               with 0.
  2   START   TXA              ; Transfer contents
                               of X to A
  3           STA     $400,X   ; Store value in ad-
                               dress $400 indexed
                               by X
  4           INX              ; Increment X
  5           CMP     #254     ; Compare A with
                               254
  6           BNE     START    ; If A is not equal to
                               254 then go back
                               to line labeled
                               START
  7           RTS
```

As you can see, it is unnecessary to always use all fields. Many programmers do not use the comment field at all, while others use

several lines for nothing but comments. (Using the comment field is equivalent to using REM statements in BASIC.) However, just about every editor arranges its fields as they are above.

Editor/Assembler program writers have an idea of how to best put assembler packages together, and so there are differences that creep in using one package or another. Perhaps the biggest is in the use of "Pseudo-Opcodes." Pseudo-opcodes are instructions that are never compiled into object code. Instead, they tell the editor to do something with the code. Often, they are called "directives" to differentiate them from opcode "instructions." For example, ORG and EQU are two commonly used directives for defining the load location of a program and defining addresses with labels. For example the following is a common label definition you will see in listings:

```
HOME EQU $E544
```

and instead of keying in,

```
JSR $E544
```

the programmer can enter,

```
JSR HOME
```

If you plan to use an assembler package not discussed in the next three chapters, keep in mind what we have discussed here. In that way, if the documentation for your assembler is unclear, you can at least expect to find the fields we discussed above. Go back over your documentation and then go ahead to Chapter 8 and try out some of the example programs. If you can get them assembled and running, you can learn more about how to use your assembler as we go along comparing what's in the book with your documentation. You might also want to take a look at the way the Commodore assembler and the Merlin Assembler assemblers are used to help you understand your own.

USING THE KIDS' ASSEMBLER

First off, if you have either the Merlin Assembler or the Commodore 64 Macro Assembler Development System skip this chapter, and go on to the chapters covering the assembler you have. In fact, if you have any other assembler you know how to work, use it instead of this one. However, if you have been confused by your present assembler, the Kids' Assembler may help you get started. Most importantly, by keying in this assembler, you'll learn about what an assembler does. In Appendix A, there's a full blown version of the Kids' Assembler. The one in this section has fewer opcodes and it runs faster than the big one in Appendix A. I also included a routine to save source code in this chapter. It doesn't save the source code in a way you can re-use it, but you can read it. Also, I've broken it down into sections so that you can do a piece at a time.

Once you key this in and get all the typing errors out, be sure and make a back-up of it on a separate tape or disk!!! There are two different ending routines beginning in Line 760. The first one is for disk users and the second is for tape users. Use only one or the other. (If you have a disk system and a cassette, just use the disk version.) By the way, feel free to modify the program any way you want.

Okay, we're all set; so let's key in the first 15 lines and then see what they do.

Kids' Assembler : C-64

```
10 POKE 53281,1 : POKE 53280,1 : PRINT CHR$(144)
20 GOSUB 4000
30 X=0
40 READ A : IF A=255 THEN 60
50 READ B$ : READ C : X=X+1 : GOTO 40
60 DIM DEC%(X),OPCODE$(X),BYTE%(X)
70 DIM AD(255),S$(255),C$(255)
80 ER=X-1
90 RESTORE
100 FOR I=0 TO X-1 : READ DEC%(I) : READ
OPCODE$(I) : READ BYTE%(I)
110 NEXT I
```

```
120 PRINT CHR$(146);CHR$(147)
130 PRINT "ADRS";
TAB(10);"OPCODE";TAB(25);"OPERAND"
140 FOR X=1 TO 40 : PRINT CHR$(114); : NEXT
150 PRINT
```

First the program sets the background and border colors to white and the characters to black in Line 10. This is simply to make it easy for people using CRT monitors or black and white TV sets to see the screen. If you like the blue colors, leave the line out or POKE in your own colors. Line 20 goes to a header subroutine to give you something to look at while the array is being loaded between Lines 40-110. The variable ER in line 80 is used in the ERror trapping routine further on in the program. Line 30 just initializes the X variable. (This is optional but a generally good habit.) Line 120 turns off the inverse mode and clears the screen. Finally, Lines 120-150 make your editor header showing the opcode and operand fields along with the addresses where everything is going. It does not have a label field, comment field or lines numbers. The line numbers are replaced by the addresses to give you a clearer idea of what's going on with the assembler. The variables and arrays defined are:

X,I	Counter Variables
A,B$ & C	Data viewer variables
DEC%( )	Decimal value of machine opcode
OPCODE$( )	Nmemonic opcode
BYTE%( )	Number of bytes used by instruction
AD,S$,C$	Array variables for source code

This next block of code sets the starting address and asks for the opcode.

```
160 REM ****************************
170 REM SET ADDRESS AND INPUT OPCODE
180 REM ****************************
190 SA=0 : PRINT "PRESS {RETURN} TO DEFAULT
TO 49152"
200 N=0
```

```
210 INPUT "STARTING ADDR";SA : IF SA = 0 THEN
SA = 49152
220 BA = SA
230 PRINT SA;TAB(10)
240 INPUT OC$ : IF OC$ = "Q" THEN 760
250 C = 0
260 IF OC$ = OPCODE$(C) THEN D% = DEC%(C) :
B% = BYTE%(C) : GOTO 290
270 C = C + 1 : IF C >ER THEN PRINT
TAB(10);CHR$(18);"ERROR";CHR$(146) : GOTO 230
280 GOTO 260
290 IF B% = 1 THEN POKE SA,D% : SA = SA + 1
300 IF B% = 1 THEN S$(N) = OC$ : AD(N) = SA-1 :
N = N + 1 : GOTO 230
```

The first thing this block does is to set the starting address. It defaults to 49152 ($C000) since that's a clear area of RAM. A lot of programmers use the cassette buffer at 828 ($033C) since it is free for disk users. Line 220 defines the constant BA to be the same as SA (starting address) since SA is used as an address variable, and we'll need the starting address later in the program. Lines 240 to 280 evaluate the opcode entered by the INPUT statement at Line 240. It searches the arrays for a match in line 260 and gets the decimal value of the machine code and the number of bytes the instruction will use. Line 290 sees if the opcode only uses one byte, and if it does then it enters the machine opcode in the address and returns to line 230 for another opcode. Line 300 stores the source code information. The variables are:

SA	Variable address
BA	Constant to store beginning address
OC$	Opcode entered
D%	Decimal value of current opcode
B%	Number of bytes used by current opcode
C	Counter variable

Now that we have the opcode entered, let's bring on the operand.

```
310 REM **************
320 REM ENTER OPERAND
330 REM **************
340 PRINT TAB(25); : PRINT CHR$(145); : INPUT OPR$
350 AD(N) = SA : S$(N) = OC$ : C$(N) = OPR$ : N = N + 1
360 IF LEFT$(OPR$,1) < > "$" THEN
OPER = VAL(OPR$)
370 IF LEFT$(OPR$,1) = "$" THEN GOSUB 490
380 IF OPER > 65535 THEN GOSUB 630 : OPER = 0:
GOTO 340
390 IF OC$ = "BNE" OR OC$ = "BEQ" THEN GOSUB
700
400 IF OPER >255 AND B% < 3 THEN GOSUB 560:
OPER = 0 : GOTO 340
410 IF OPER > 255 THEN GOSUB 640
```

First, in Line 340, the INPUT line is tabbed over and up to the OPERAND field so you can see what you're supposed to enter. Next, Lines 360-410 check out the operand for all sorts of conditions:

360 If this is a decimal stick it in the variable OPER
370 If this is a hexadecimal number, go convert it
380 If this number is greater than 65535 go to the error routine and have the programmer try again
390 If there's a conditional branch in the opcode go find the branch offset
400 If the value is greater than 255 and it's only a 2 byte opcode, go to the error routine and have the programmer try again.
410 If the value is over 255, go get it rearranged into the lo-byte / high-byte values.

The new variables introduced are:

OPR$	Hex or decimal operand in string
OPER	Numeric variable of operand

Once everything is all checked out and conversions are made after the operand is entered, the code is immediately compiled in the next routine.

```
420 REM ************
430 REM COMPILE CODE
440 REM ************
450 IF B% = 2 THEN POKE SA,D% : SA = SA + 1
460 IF B% = 2 THEN POKE SA,OPER : SA = SA + 1 :
OPER = 0 : GOTO 230
470 POKE SA,D%: SA = SA + 1
480 POKE SA,LB : SA = SA + 1 : POKE SA,HB :
SA = SA + 1 : OPER = 0 : GOTO 230
```

The compile block shows you just what's going on with an assembler. It merely POKEs in machine code values for the mnemonic opcodes and operands and keeps the addresses straight. Lines 450-460 check for two byte opcodes, and then pops in their values at the next two addresses, resets the opcode value, increments the address, and then goes back to get the next opcode. Lines 470and 480 do the same thing for 3 byte opcodes using the LB (low byte) and HB (high byte) variable supplied in a subroutine further on in the program. (Line 410 in the previous block branched to the subroutine for the LB and HB variables.) The new variables, LB and HB will be discussed further on in the block where they are defined.

At this point, go have a good stiff shot of root beer. (I did.) We've actually gone through the entire assembly process! From this point on, we will see the subroutines, DATA statements and ending block. They're crucial to the program, but the heart of the program is already keyed in.

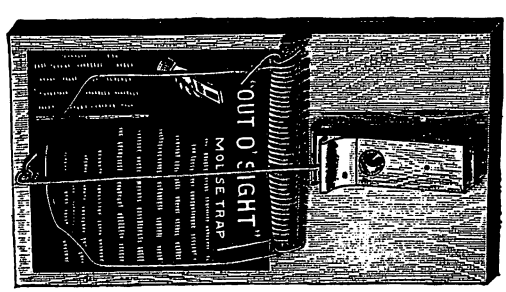
Now we're ready to start on the subroutines. The first one is a very good little one to make HEX-DECIMAL conversions in any program. (HINT# $FF : If you want to write your own monitor, this subroutine would be handy.)

```
490 REM **********************
500 REM CONVERT HEX TO DECIMAL
510 REM **********************
520 H$ = MID$(OPER$,2)
530 FOR L = 1 TO LEN(H$) : HD = ASC(MID$(H$,L,1))
540 OPER = OPER*16 + HD - 48 + ((HD > 57)*7)
550 NEXT L: RETURN
```

The heart of this subroutine is in Line 540. The loop between 530-550 converts the hexadecimal value in H$ to a decimal number that can be POKEd into memory in the compile subroutine. Line 520 simply strips the '$' off OPER$ and stores the substrong in H$.

Next, we come to the double error trap for values over 255 ($FF) entered with 2 byte opcodes and any operand value over 65535 ($FFFF).

```
560 REM **********
570 REM ERROR TRAP
580 REM **********
590 PRINT CHR$(18);"ERROR-MUST BE LESS THAN
256"
600 FOR W = 1 TO 400 : NEXT W : PRINTCHR$(146); :
PRINT CHR$(145)
610 FOR X = 1 TO 27 : PRINTCHR$(32); : NEXT
620 PRINT CHR$(157);CHR$(157);CHR$(145) :RETURN
630 PRINT CHR$(18);"VALUE OVER 65535
($FFFF)";CHR$(146) : RETURN
```



Error Trap

There's nothing fancy about this subroutine. It simply pops an error message if you're over 255 (a little spiffy, I admit) and does the same thing in a 1 line subroutine in 630 if your operand exceeds 65535.

Now the next subroutine is another one you could use in a monitor program. It takes any number over 255 and splits it into high and low bytes.

```
640 REM ************************
650 REM CONVERT TO 2 BYTE NUMBER
660 REM ************************
670 LB = OPER-INT(OPER/256)*256
680 HB = OPER – INT(OPER/256)
690 RETURN
```

The routines in 670 and 680 do all the work. They're simple but vital little formulas. See how these variables are compiled in Line 480. They created the following variables:

LB	Low byte - first address
HB	High byte - second address

Now this next subroutine determines the operand value by comparing two addresses. The value is the "branch offset" sending the program branching forward or backwards.

```
700 REM *************
710 REM BRANCH OFFSET
720 REM *************
730 IF SA  > OPER THEN OPER = 254-(SA-OPER)
740 IF SA  < OPER THEN OPER = (OPER-SA)-2
750 RETURN
```

Notice how the value has to be calculated differently depending on whether the address value (SA) is greater or lesser than the branch address (OPER).

At this point, take a good look at your system. If you use a cassette to store your programs, skip this and go to the next block. If

you have a disk, this is the block you want. It gives you the options of saving your program, ending or going back to the beginning. The most valuable part of the routine is in Lines 890-970. It contains the save-to-disk-as-a-program routine for your OBJECT CODE! Lines 980-1040 save the source code as a SEQ file.

Disk Version Only

```
760 REM **************
770 REM ENDING ROUTINE
780 REM **************
790 NB = SA-BA
800 PRINT CHR$(147)
810 FOR X=1 TO 5 : PRINT : NEXT
820 INPUT"SAVE PROGRAM(Y/N)";AN$
830 IF AN$="Y" THEN 890
840 PRINT : PRINT : PRINT "PROGRAM
IS";NB;"BYTES LONG"
850 PRINT "TO EXECUTE 'SYS"';BA : PRINT
860 INPUT "(B)EGIN AGAIN OR (E)ND";DE$
870 IF DE$="B" THEN 120
880 PRINT : PRINT"END" : END
890 PRINT CHR$(147) : FOR X=1 TO 5 : PRINT :
NEXT
900 LB=BA-INT(BA/256)*256 : HB=INT(BA/256)
910 INPUT "ENTER FILE
NAME";NW$:NF$=NW$:NF$="0:"+NF$+STR
$(BA) +",P,W"
920 OPEN2,8,2,NF$
930 PRINT#2,CHR$(LB)+CHR$(HB)
940 FOR X=BA TO SA-1: OC=PEEK(X)
950 PRINT#2,CHR$(OC)
960 NEXT X
970 CLOSE2
980 NF$=""
990 NF$="0:" + NW$+",S,W"
```

```
1000 OPEN 9,8,9,NF$
1010 FOR V = 0 TO N-1
1020 PRINT#9,AD(V),S$(V),C$(V)
1030 NEXT V
1040 CLOSE9
1050 GOTO 840
```

There's nothing fancy about the save, end or begin branches. However, our constant, BA is used to find the number of bytes in the program by subtracting it from SA, the last address entered + 1. The save-to-disk routine from 890-970 is one I got from Guy Grotke's book, *The Intermediate Commodore 64*. It stores the beginning address of your machine language program as part of the file in line 910. Therefore, when you LOAD "FILENAME",8,1 the program knows where to go. To help you remember that, I added the starting address to the file name. Therefore, when you save a machine file, the name includes its load address and all you have to do once the program is loaded is to SYS that address. The new variables in this block are :

NB	Number of bytes in program
X	Counter variable
AN$, DE$	Strings for INPUT branches
NF$	Name of file to write to disk
OC	Decimal value of opcodes and operands

If you just finished keying in the above block of the ENDING ROUTINE jump (JMP in mnemonic opcode) to the block, OP-CODE DATA. This following block is a *repeat* of the ENDING ROUTINE except it is for saving a sequential file of your program to tape instead of a PRG file to disk.

Tape Version Only

```
760 REM **************
770 REM ENDING ROUTINE
780 REM **************
790 NB = SA-BA
800 PRINT CHR$(147)
```

```
810 FOR X=1 TO 5 : PRINT : NEXT
820 INPUT"SAVE PROGRAM(Y/N)";AN$
830 IF AN$="Y" THEN 890
840 PRINT: PRINT: PRINT "PROGRAM IS";NB;"BYTES
LONG"
850 PRINT "TO EXECUTE 'SYS'";BA : PRINT
860 INPUT "(B)EGIN AGAIN OR (E)ND";DE$
870 IF DE$="B" THEN 120
880 PRINT : PRINT"END" : END
890 PRINT CHR$(147) : FOR X=1 TO 5 : PRINT : NEXT
900 REM *** TAPE SAVE ***
910 INPUT "ENTER FILE NAME";NW$:NF$=NW$
920 OPEN21,1,1,NF$
930 PRINT#21,BA
940 FOR X=BA TO SA-1: OC=PEEK(X)
950 PRINT#21,OC
960 NEXT X
970 CLOSE21
980 NF$=""
990 NF$=NW$+".S"
1000 OPEN22,1,1,NF$
1010 FOR V=0 TO N-1
1020 PRINT#22,AD(V),S$(V),C$(V)
1030 NEXT V
1040 CLOSE22
1050 GOTO 840
```

The good news is that you can save machine language programs to tape, but the bad news is they are saved as SEQ files instead of PRG files. That means we will have to have a special "Loader" program for you tape users. We'll look at that later. For now, let's see what the above block does. First of all, there are decision branches in lines 820-880. These just prompt you with questions about saving your program and whether you want to end or start over. Line 790 finds the number of bytes (NB) in your program to be used both in telling you how much memory you used and as a variable to be stored on your tape file. Line 920 begins the storage sequence of your program. Basically, this section OPENs a tape file, first stores the number of bytes in your program (Line 930), and then it PEEKs

at your machine language program in memory and stores the values on your tape. The loop in Lines 940-960 does this. The following variables were introduced in this block:

NB	Number of bytes in program
X	Counter variable
AN$, DE$	Strings for INPUT branches
NF$	Name of file to write to tape
OC	Decimal value of opcodes and operands

Now this next section will have to be done *very* carefully. It has all of the information your assembler will use. Each DATA statement uses a separate line containing the following information:

Decimal value of opcode
Mnemonic for opcode
Number of bytes used by opcode and mode

The reason I put every set of DATA statements in a separate line was to help you debug typing errors and to see the relationship between machine opcode (the first number), mnemonic opcode (the string), and the number of bytes used by an operation (the second number). Also, you will get a preview of the special conventions this assembler uses in mnemonic opcodes. In most discussions of machine and assembly language programming, the machine opcode is given in hexadecimal values, but since your assembler is written in BASIC, it needs the decimal values to 'compile' the code into memory. OK, now take a deep breath and go ahead and key in this code.

```
2000 REM ***********
2010 REM OPCODE DATA
2020 REM ***********
2030 DATA 24,CLC,1
2040 DATA 32,JSR,3
2050 DATA 56,SEC,1
2060 DATA 73,EOR#,2
2070 DATA 76,JMP,3
2080 DATA 77,EOR,3
2090 DATA 96,RTS,1
```

```
2100 DATA 105,ADC#,2
2110 DATA 108,(JMP),3
2120 DATA 109,ADC,3
2130 DATA 121,ADC-Y,3
2140 DATA 125,ADC-X,3
2150 DATA 129,(STA-X),2
2160 DATA 133,STA-Z,2
2170 DATA 134,STX-Z,2
2180 DATA 136,DEY,1
2190 DATA 138,TXA,1
2200 DATA 140,STY,3
2210 DATA 141,STA,3
2220 DATA 142,STX,3
2230 DATA 145,(STA-Y),2
2240 DATA 148,STY-X,2
2250 DATA 152,TYA,1
2260 DATA 157,STA-X,3
2270 DATA 153,STA-Y,3
2280 DATA 154,TXS,1
2290 DATA 160,LDY#,2
2300 DATA 161,(LDA-X),2
2310 DATA 162,LDX#,2
2320 DATA 164,LDY-Z,2
2330 DATA 165,LDA-Z,2
2340 DATA 166,LDX-Z,2
2350 DATA 168,TAY,1
2360 DATA 169,LDA#,2
2370 DATA 170,TAX,1
2380 DATA 172,LDY,3
2390 DATA 173,LDA,3
2400 DATA 174,LDX,3
2410 DATA 177,(LDA-Y),2
2420 DATA 185,LDA-Y,3
2430 DATA 186,TSX,1
2440 DATA 188,LDA-Y,3
2450 DATA 189,LDA-X,3
2460 DATA 190,LDX-Y,3
2470 DATA 192,CPY#,2
2480 DATA 193,(CMP-X),2
2490 DATA 196,CPY-Z,2
```

There's a lot of DATA in here!

```
2500 DATA 197,CMP-Z,2
2510 DATA 198,DEC-Z,2
2520 DATA 200,INY,1
2530 DATA 201,CMP#,2
2540 DATA 202,DEX,1
2550 DATA 204,CPY,3
2560 DATA 205,CMP,3
2570 DATA 26,DEC,3
2580 DATA 208,BNE,2
2590 DATA 221,CMP-X,3
2600 DATA 222,DEC-X,3
2610 DATA 224,CPX#,2
2620 DATA 230,INC-Z,2
2630 DATA 232,INX,1
2640 DATA 233,SBC#,2
2650 DATA 234,NOP,1
2660 DATA 236,CPX,3
2670 DATA 237,SBC,3
2680 DATA 238,INC,3
2690 DATA 240,BEQ,2
2700 DATA 249,SBC-Y,3
2710 DATA 253,SBC-X,3
2720 DATA 254,INC-X,3
2730 REM ************************
2740 REM ADD ADDITIONAL DATA HERE
2750 REM ************************
2760 DATA 255
```

It is important to have line 2760 be the LAST DATA statement entered. When your program READs 255 as a value, it knows that it is at the end of the DATA. Therefore, if you get ambitious and add additional opcode, move Line 2760 to the end of your new DATA statements. In the full program in Appendix A, you will find all of the DATA values for additional opcode. (Be careful in adding BCC, BCS, BMI, BVC and BVS. When used, their operands will have to be sent to the branch offset subroutine in beginning in line 700. All you have to do is to add lines like 390 to initiate the branch. Use line numbers 391-399 for branch initiators with these branch opcodes

not included in this simplified version of the Kids' Assembler. The full assembler in Appendix A takes care of all of this for you, of course.)

Finally, we get to the commercial in this last block. The header was included so that if you give a copy of the program to your friend, he or she will know where to find the documentation on how to use the assembler. If you ever had a program without the documentation, you know how frustrating it is to use the program. This is especially true with complex utilities like assemblers. Therefore, this last block is important. (Also, it's a get-rich-quick scheme to sell more books!)

```
4000 REM ******
4010 REM HEADER
4020 REM ******
4030 PRINT CHR$(147)
4040 CR$ = "(C) COPYRIGHT 1984" : NM$ = "BY
WILLIAM B. SANDERS"
4050 BK$ = "ASSEMBLY LANGUAGE FOR KIDS"
:CM$ = "COMMODORE 64"
4060 IS$ = "SEE" : F$ = "FOR DOCUMENTATION"
4070 H = 20-LEN(CR$)/2 : PRINT TAB(H);CR$
4080 H = 20-LEN(NM$)/2 : PRINT TAB(H);NM$
4090 PRINT: H = 20-LEN(IS$)/2 : PRINT TAB(H);IS$ :
PRINT
4100 H = 20-LEN(BK$)/2 : PRINT TAB(H);BK$
:H = 20-LEN(CM$)/2: PRINT TAB(H);CM$
4110 H = 20-LEN(NM$)/2 : PRINT TAB(H);NM$ : PRINT
4120 H = 20-LEN(F$)/2 : PRINT TAB(H);F$
4130 LD$ = "LOADING ARRAY" : FOR X = 1 TO 10 :
PRINT : NEXT : H = 20-LEN(LD$)/2
4140 PRINT TAB(H);CHR$(18);LD$
4150 RETURN
```

Phewwww! That sucker was long. Well, how many kids do you know who wrote their own assemblers? If you did all that work, you won't understand everything about assemblers and assembly language, but you'll be ahead of those who haven't. Congratulations! (Only real programmers write their own assemblers!!!) Save

the program under the name "KIDS ASSEMBLER 1." We'll call the big one in Appendix A, "KIDS ASSEMBLER 2."

CREATING AND SAVING PROGRAMS ON THE KIDS' ASSEMBLER

Once you get all the typing errors out of your assembler, you're all set to crank it up. REMEMBER to make a back-up copy or two on separate disks or tapes. If you've done that, then LOAD "KIDS ASSEMBLER 1" and then enter RUN.

The first thing you will see when you RUN the program is the following header:

```
(C) COPYRIGHT 1984
BY WILLIAM B. SANDERS

SEE

ASSEMBLY LANGUAGE FOR KIDS:
COMMODORE 64
BY WILLIAM B. SANDERS

FOR DOCUMENTATION

{LOADING ARRAY}
```

The message stays there until the array is loaded and lets you know something is happening. After a few seconds your editor will pop up. It looks like the following:

```
ADRS        OPCODE      OPERAND
-------------------------------------------
PRESS {RETURN} TO DEFAULT TO 49152
STARTING ADDR?
```

At this point you are expected to enter a starting address for your program. If you just press the RETURN key, your program will automatically begin compiling at 49152 ($C000). Generally, this

area of RAM is free for machine language programs and I like to use it. If you do not use a cassette tape, the cassette buffer at 828 ($33C) is also a good place to use. Later on we'll discuss the various place where you can put your code. For now, just press the RETURN key. As soon as you do, your screen looks like this:

```
ADRS          OPCODE        OPERAND
-------------------------------------------
PRESS {RETURN} TO DEFAULT TO 49152
STARTING ADDR?
49152         ?
```

The prompt under the OPCODE field is waiting for you to enter an opcode. (What else?) Enter JSR and press RETURN. Now your screen looks like this:

```
ADRS          OPCODE        OPERAND
-------------------------------------------
PRESS {RETURN} TO DEFAULT TO 49152
STARTING ADDR?
49152         ? JSR         ?
```

The prompt has jumped to the OPERAND field awaiting an operand. Now, you can either enter the operand as a decimal or hexadecimal number. If you choose to enter a decimal number, just key in the number. For hexadecimal numbers, though, first put in a dollar sign ($) and then the number. To get started, enter the value 58692 or $E544 and press RETURN. Both numbers are the same and have the same effect. The hexadecimal to decimal subroutine automatically changes $E544 to the decimal value 58692. It's a good habit to start thinking in terms of hexadecimal. After you've done that, you screen will appears as:

```
ADRS          OPCODE        OPERAND
-------------------------------------------
PRESS {RETURN} TO DEFAULT TO 49152
STARTING ADDR?
49152         ? JSR         ? $E544
49155         ?
```

You've successfully entered a line of assembly code, and your assembler is waiting for the next line. That's all there is to it! Some opcodes have no operands, but for the most part, you just enter the opcode and the operand. The last line of your code should be RTS to return control of your computer to BASIC once the program is executed with a SYS command. It's not a very powerful assembler, but it's easy to use and will catch many common errors that beginners make.

You may be wondering about the address field on the left side of your screen. The first number was 49152 and the second is 49155. Shouldn't it be 49153? What the assembler is doing is showing you how much memory each opcode and operand is using. In the first line you used three addresses or bytes. One byte was used for the opcode and two bytes were used for the operand. Thus, addresses 49152, 49153 and 49154 have been used, and the next available address is 49155. This will help you see where your code is actually going.

OK, let's end the program with RTS and RETURN. Now your screen shows the following:

```
ADRS        OPCODE      OPERAND
-------------------------------------

PRESS {RETURN} TO DEFAULT TO 49152
STARTING ADDR?
49152       ? JSR       ? $E544
49155       ? RTS
49156       ?
```

Since the RTS opcode used only a single byte and requires no operand, you are now at address 49156. To end your work in the editor enter 'Q' for 'quit' and press RETURN. You screen will clear and you will be prompted with the following message:

```
SAVE PROGRAM(Y/N)?
```

Enter 'Y' and press RETURN and you will see the next message:

```
ENTER FILE NAME?
```

Enter the file name CS (for Clear Screen, since that's what the program does) and press RETURN. Now you will see:

```
ENTER FILE NAME? CS

PROGRAM IS 4 BYTES LONG
TO EXECUTE 'SYS' 49152
(B)EGIN AGAIN OR (E)ND?
```

If you press 'B' you will be immediately returned to the editor. Since we've already seen the editor, press 'E' and RETURN to end our little session with the Kids' Assembler. Your screen will look like this now:

```
ENTER FILE NAME? CS

PROGRAM IS 4 BYTES LONG
TO EXECUTE 'SYS' 49152
(B)EGIN AGAIN OR (E)ND? {Press 'E'}

END

READY.
```

You're back in BASIC control of your computer. If you enter,

```
SYS 49152
```

and press RETURN your machine language program will execute. Go ahead and do it to see what happens.

Now here's the really neat part about machine language. You have two programs in memory at the same time. The assembler is still in memory along with your little machine language program. Just enter RUN {RETURN} to execute your assembler. Since your BASIC program begins way down at 2048 ($800) and your machine program is way up at 49152 ($C000), they won't conflict. (Go tell your mother about that.)

SPECIAL CONVENTIONS IN OPCODES

Perhaps the biggest single problem with the Kids' Assembler is its use of some non-standard opcodes notations. On the one hand, these conventions were used to help you understand exactly what a mnemonic opcode is in relationship to a machine language opcode. On the other hand, it is a heck of a lot easier to write a relatively short assembler in BASIC using the conventions I did! (Now you know the awful truth.)

As we will see in Chapter 8, the 651Ø has several different addressing modes. On most assemblers, the modes are determined in the operand field. For example the following shows the instructions for loading the accumulator with a 5 in the immediate mode and absolute mode on most assemblers:

```
OPCODE    OPERAND
LDA       #5 ◄-Immediate mode
LDA        5 ◄-Absolute mode
```

The assembler can tell the first LDA instruction is in the immediate mode since there is a pound sign (#) before the 5 in the operand field. The second LDA instruction, however, is in the absolute mode. The first LDA tells the computer to load the value 5 into the accumulator. The second LDA, in the absolute mode, tells the computer to load the value from address 5 into the accumulator. What actually happens when the code is turned into machine language is that the machine opcode for the first LDA is stored as $A9 (169 decimal) and the second LDA is stored as $AD (173) decimal. Since the mnemonic opcode can be translated directly into a machine opcode, by having the addressing mode as part of the mnemonic opcode, you can better see the translation going on. Therefore, in the Kids' Assembler, you put the addressing mode as part of your opcode. Opcodes for the absolute, relative, and implied mode are exactly the same as on standard assemblers, but for other modes the following conventions are used. (Don't worry about all the details of addressing modes now. Later on in the book, we'll tackle each one separately. Just take a look at the different conventions used.)

```
OPCODE    OPERAND
LDA#      5 ◄- Immediate mode
LDA        5 ◄- Absolute mode
```

As you can see, the absolute mode on the Kids' Assembler is the same as the standard ones. However, the pound sign (#) has been moved from the OPERAND field to the OPCODE field in the immediate mode. The following is a full list of conventions you can refer back to later when we cover the various addressing modes.

```
LDA       Absolute mode (standard)
LDA#      Immediate mode
TXA       Implied (standard)
BNE       Relative mode (standard)
LDA-Z     Zero page mode
(JMP)     Indirect
LDA-X     Indexed
(LDA-X)   Indexed indirect
(LDA-Y)   Indirect indexed
```

Basically, the differences lay in where you put the special symbols. I tried to make them consistent with the standard ones, and simply place them with the opcode instead of the operand.

LOADING AND EXECUTING PROGRAMS

One of the best things about this assembler is the ease with which you can load and execute machine language programs. From disk all you do is to enter:

```
LOAD "PROGRAM NAME 49152",8,1
```

You then SYS the numeric value attached to the file name your program will execute. The program is automatically loaded to the start address from which you saved the program. Also it will not disturb a BASIC program in memory. (Well almost, anyway.)

Getting your programs loaded from tape takes a special loader program. That's because it's stored as a SEQ file and you have to first read the file and then POKE the whole thing into memory. (The best way to solve this problem is to buy a disk drive.) It's no problem though with the following cassette machine language loader program.

```
10 PRINT CHR$(147)
20 INPUT "NAME OF FILE TO LOAD "; NF$
30 INPUT "ADDRESS TO LOAD"; SA
40 OPEN1,1,0,NF$
50 INPUT#1,NB
60 FOR X = SA TO SA + (NB-1)
70 INPUT#1,CD
80 POKE SA,CD
90 NEXT X
100 CLOSE1
```

Then just SYS the beginning address you entered when prompted ADDRESS TO LOAD?

READING SOURCE FILES

Since a SEQ source file is saved with the object file in this version of the Kids' Assembler, you will need a program to read your source code. The first of the following two is for disk and the second is for tape:

SOURCE CODE READER DISK

```
10 PRINT CHR$(147)
20 INPUT "FILENAME ";NF$
30 NF$ = "0:" + NF$ + ",S,R"
40 OPEN9,8,9,NF$
50 INPUT#9,A$
60 PRINT A$
70 IF ST = 0 THEN 50
80 CLOSE9
```

SOURCES CODE READER TAPE

```
10 PRINT CHR$(147) : X = 0
20 INPUT "NAME OF SOURCE FILE ";NF$ :
   NF$ = NF$ + ".S"
30 OPEN22,1,0,NF$
40 INPUT#22,A$,B$,C$
50 PRINT A$,B$,C$
60 A$ = "":B$ = "":C$ = ""
70 IF ST = 0 THEN 40
80 CLOSE22
```

The above programs will print your sources to the screen so that you can see how you programmed your object code. It cannot, unfortunately, be loaded into your editor and reused.

SOME EXAMPLES

Before going on the Chapter 3, crank up your assembler for some test runs. This will give you further checks on typos in your program and show you that assembly language isn't impossible.

ERROR TESTS

```
ADRS   OPCODE     OPERAND
49152  ? XYX
       {ERROR}

49152  ? LDA#     ? 500 ◄- Enter 0 on second try
{ERROR - MUST BE LESS THAN 256}
49154  ? LDA      ? $FFFFA
{VALUE OVER 65535 ($FFFF)} ? 828
49157  ? Q
```

The above program just tested the error traps built into your assembler. There's nothing worth saving; so just go back to the beginning.

BACKGROUND, BORDER AND CHARACTER COLORS

```
ADRS   OPCODE    OPERAND
49152  ? JSR     ? $E544
49155  ? LDA#    ? 0
49157  ? STA     ? $D021
49160  ? LDA#    ? 4
49162  ? STA     ? $D020
49165  ? LDA#    ? 5
49167  ? JSR     ? $E716
49170  ? RTS
49171  ? Q
```

When you SYS 49152 after you've exited the assembler, your background will turn black, your border purple and your characters white. If it worked, go on to Chapter 5. If it didn't, try it again and make sure everything looks as it does above. If it still doesn't work, check your BASIC assembler program, especially the DATA in the opcode block.

CHAPTER 3

THE MERLIN 64 ASSEMBLER

Using the Editor/Assembler

This is the best assembler I've seen for the Commodore 64. It only works with a disk system; so if you have cassette storage, you'll have to wait until you have a disk drive to use this one.

First of all, make a back-up copy of the Merlin Assembler and put the original in a safe place. Using the back-up master, enter the following:

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

When it's loaded, enter RUN, press RETURN, and patiently wait while the program is cranked up. (It takes a while.) Remove your back-up master and put in your work disk. This should be a formatted disk that you can afford to destroy. It is possible to accidentally write a machine code program that will cream your disk. (I do it all the time.) In fact, it's a good idea to have two work disks; one to keep in the drive while you're experimenting, and one on which to save your source and object code.

The program starts in the "Executive Mode." You will see the following on your screen:

```
                    {MERLIN}

                  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
G :Run program
X :Disk command
Q :Quit
                  Source: $0A00,$0A0
Drive: 8

%
```

From the Executive Mode, you can do a lot of house-keeping. The first thing to do is to initialize your work disk.

Press X

The prompt will then show:

%Command:

Just enter "i" and press RETURN.

To see what's on the disk, press C for "Catalog." Your file directory will appear on the screen. There's nothing more to do in the

Executive Mode for the time being; so hit RETURN and enter the Editor/Assembler by pressing 'E'. Your screen will look like this:

```
Editor
:
```

This is called the Command Mode. From here, you want to get into the Add/Insert Mode. This is where you begin writing your assembly language code. You can recognize it by the colon (:) prompt. To get to the point where you can start writing assembly programs, enter;

```
:A {RETURN}
```

As soon as you hit RETURN after entering the 'A' you're in the "Add Mode." It is in this mode where you will be doing most of your work. The number '1' will pop up, representing your first line number.

```
Editor
:A
 1
```

As in most assemblers, Merlin has four fields:

```
LABEL     OPCODE   OPERAND COMMENT
```

Each time you hit the space bar, you go to the next field. For the most part, the OPCODE and OPERAND fields are the most important. When we get to branches, the LABEL field becomes very important, and as we will see in later chapters, it is used extensively for defining special locations and addresses as well. In our examples in this chapter, we will show you how to use all four fields. To get started, though, just try the following little program using the OPCODE and OPERAND fields:

```
LN#  LABEL    OPCODE   OPERAND COMMENT
1             JSR      $E544
2             RTS
```

To enter the routine, enter the Add Mode, and as soon a the '1' appears do the following:

Step 1: Press the space bar to jump to the OPCODE field and enter JSR.
Step 2: Press the space bar again to jump to the OPERAND field and enter $E544. Now press the RETURN key to go to the next line.
Step 3: When the '2' appears, press the space bar to jump to the OPCODE field and enter RTS.
Step 4: Press RETURN *twice.* The first RETURN will take you to the next line number, and the second RETURN will take you out of the Add Mode and back into the Command Mode. Pressing RETURN twice without pressing any other key will always take you out of the Add Mode.

The following shows how your screen should look now:

```
Editor
:A
1           JSR       $E544
2           RTS
3 ◄–
:
```

At this point, all the information you need for your assembly language program is in the editor. In order to get in into a form you can execute as a program, you must assemble it. To do that, from the Command Mode, you enter,

```
:ASM
```

and press RETURN. You will be prompted,

```
:ASM
Update source (Y/N)?
```

You do not want to update the source code; so you press 'N', and your code will be assembled. Your screen now looks as follows:

```
 3 ◄-
:ASM
Update source (Y/N)?
Assembling
8000:       20 44 E5  1         JSR       $E54
4
8003:                 2         RTS

—End assembly, 4 bytes, Errors: 0
Symbol table - alphabetic order:
Symbol table - numerical order:
:
```

If that's what your screen looks like, you've just successfully assembled your first program on Merlin. Next we will want to save the program to disk, get out of Merlin, and then see if the program runs. Before we do that, though, make a note of the first number in the assembled code. It is the hexadecimal value $8000, shown on your screen simply as 8000. This is the address where your *object code* (your assembled machine language program) will load. It's decimal value is 32768, and once your program is loaded, SYS 32768 will be used to execute it.

Now press 'Q' for quit and you will be returned to the Executive Mode. The '%' prompt appears enter,

```
%S
```

Your screen will show:

```
%Save:
```

Be sure your work disk and not your master is in the drive and enter a file name ('clear' is a good name since that's what the program will do - clear your screen), and press RETURN. Your screen will show:

```
%Save:clear
Saving clear.s
```

After saving the file, your screen will clear and you will be back in the Executive Mode. Now, you have *not* saved anything that will execute. You have just saved the source code. (Merlin put the '.s' on the end of your file name to distinguish it from the object code.) This file has all the information you entered in the editor, but it did not save the assembled code. To save what you assembled, called the "object code," enter 'O' after the '%' prompt. (Make sure that's an 'Oh' and not a 'Zero'.) When you do that your screen will look like this:

```
%Object:clear
```

Since you just saved the source code for a program named 'clear', Merlin assumed that the next file you want is an object code called 'clear'. You do and so just press RETURN without entering any file name.

```
%Object:clear
Saving clear.o
```

Now you have saved *both* your source code and object code. When you look at your disk's directory with the 'C' command, you will see a file named 'clear.s' and one named 'clear.o'. (Later when you're not using Merlin and have your keys set for upper case, the files will appear as 'CLEAR.S' and 'CLEAR.O'.)

Now, before we leave Merlin, take a look at the right side of your screen. There you will see the addresses with your source and object codes.

```
                Source: $0A0,$0A10
Drive:          8 Object: $8000,$8004
```

In case you forgot to note the load address of your object code after you assembled it, it is supplied here for you.

Let's get out of the Executive Mode and back to BASIC. Press 'Q' and when asked 'Do you really want to exit?', press 'Y' for "Yes." Your screen will clear and you will be notified that to re-enter Merlin use SYS 52000.

Once you're back in BASIC you can execute your program. When you entered ASM to assemble it in Merlin, it was placed in memory at location $8000 (32768). Therefore, if you SYS 32768 and press RETURN, your program will execute. Go ahead and try it. If you did everything right, your screen should clear. See how easy that was?

Use the editor to make changes

EDITING YOUR SOURCE CODE WITH MERLIN 64

To get back into the editor/assembler, just enter,

SYS 52000

and press RETURN. You'll be returned to the Executive Mode in Merlin. Now, choose 'E' to enter the ED/ASSEM. As soon as you're in the Command Mode indicated by the colon (:) prompt, press 'L' and RETURN. Your source code should be there waiting for you. The 'L' (L)isted your program; so now you know one of the first editing commands. Whenever you want to see your source code in memory, just press 'L' in the Command Mode. Now, let's take a look at the major editing commands. We'll start with a list of the most important ones, and then show you how to use them.

EDITING COMMANDS

L	Lists source code
A	Enter editor at next available line number
I#	Insert code into line number #
D#	Delete line number #
E#	Edit line number #
PORT#	Selects printer as output port #

PRTR#	Sends formatted listing to printer in port #.
F	Find a string in listing
C	Change string
M	Move lines of code
R	Move lines in one range to another section
NEW	Clears source code from memory

L(ist). We've already seen how to list a source code with L. However, when your programs get longer, you may want to stop the listing to examine a certain section. Whenever you press the space bar during a listing, you can step through the listing one line at a time. To resume normal listing, press RETURN.

A(dd). When we first started writing our program, we pressed 'A' from the command mode and got a 1. With your program in memory, press 'A' and you will be given the next available line number. Having used only 2 lines, if we now press 'A' we'll start at line 3.

I(nsert)#. This command is used to (I)nsert lines between lines. With your 2 line program in memory, enter I1 {RETURN}. You will now be in line 1. Press the space bar and enter the following:

```
ORG     $C000
```

Press RETURN twice to get back to the Command Mode and press L to list your program. Now it looks like this

```
1       ORG     $C000
2       JSR     $E544
3       RTS
```

The ORG instruction is a "pseudo-opcode." It is not assembled into object code, but instead it identifies the beginning address for your code. If you do not put an ORG in your program, it defaults to the beginning address $8000. It should be the first line of code in your programs. As you learn programming in assembly language, you will be leaving out a lot of code, and it will be necessary to insert

code between line numbers. Therefore, you will be using the (I)nsert function a good deal.

D(elete)#. This is really simple to use. You just enter D and the line or range of lines you want to get rid of and hit RETURN. Since we don't have any lines in our program to delete, let's stick some there to knock out. Do the following:

```
:A
   4 OINK
   5 BURP
   6 CRASH
```

Deleting unwanted code

After you enter line 6, hit RETURN twice to get back to the Command Mode. Lines 4, 5 and 6 are pretty worthless. List your program and you'll see them all there at the end of your listing. Now from the Command Mode, do the following:

```
: D 6
```

Press RETURN and press L to list your program. Line 6 has been (D)eleted. Now to delete a range of lines enter:

```
:D4,5
```

Hit RETURN twice and list your program again. All of those dumb lines are gone.

E(dit)#. To change a line, press E and the line number to edit from the command mode. Let's change our ORG from $C000 to $033C. Do the following:

```
:E1 {RETURN}
 1          ORG       $C000
```

Press the space bar to jump to ORG and then using the right cursor key, move over the ORG and hit the space bar again. The cursor should be right on top of the dollar sign ($) of the $CØØØ. Move one space to the right with the cursor key and replace $CØØØ with $Ø33C. Press RETURN and you're back in the Command Mode.

While in the edit mode you can (I)sert and (D)elete characters with CTRL-I and CTRL-D. To try out these editing functions, let's edit Line 2.

```
:E2
2          JSR        $E544
```

Move the cursor over the 'S' in JSR and press CTRL-D. The 'S' will be deleted. Now to get the 'S' back between the 'J' and the 'R' put the cursor over the 'R' and press CTRL-I and enter 'S'. Experiment with inserting and deleting code since you'll be doing a lot of editing in assembly language programming. Here's a few more Edit Mode CTRL commands to use:

CTRL-F Find a character
CTRL-O Like CTRL-I except it inserts control character
CTRL-P This is really a neat command for putting a line of asterisks across your screen. Use it for your header blocks. If you hit the space bar and CTRL-P, you'll get asterisks (*) on either side of your line.
STOP/RUN Gets out of the Edit mode without changing the line you've edited.
CTRL-B Jumps to beginning of line
CTRL-N Jumps to end of line
CTRL-R Restores the line to its original state
CTRL-A Erases the line from the cursor to the right

PORT# PRTR#. If you have a printer hooked up to your Commodore-64, you can send output to it using either the PORT or PRTR commands.

If you use PORT, specify it as PORT 2, PORT 4 or PORT 5. When you ASM your source code, you will be able to see all the information about it printed on paper. This is useful for debugging

programs. To get it going, from the Command Mode, enter the following:

```
:PORT4
```

Press RETURN and your output will be vectored to your printer.

The PRTR command works almost the same as PORT but it formats the output to your printer to include page breaks. Since you really won't need this command until your programs are longer (over 60 lines or so), you'll either have to enter a big program or wait until you are more advanced. Here's an example format

```
:PRTR 4 "MAY 8, 1990" 1
```

The string is your page header and the number at the end is your page number. The '4' refers to a channel to your printer port just like with the PORT command.

F(ind). When you start having longer listing, this command is really handy. From the command mode, you enter something like,

```
:F "JSR"
```

press RETURN, and all lines with JSR in it will be listed to the screen. If you enter the range of lines before the string you're searching for, just that range will be listed.

```
:F 5,9"JSR"
```

C(hange). This command is good for wholesale mistakes. For example let's say you have a program with JMP opcodes instead of JSR opcodes as you really wanted. With the change command, you can change all of the JMP's to JSR's from the Command Mode. Here's how to do it:

```
:C "JMP"JSR"
```

Take special note of how we used the quotation marks ("). There are three of them instead of four (two around each string.) When

you press RETURN, you will be asked whether you want A(ll) or S(ome) of the strings changed. If you press 'A' for all, every single instance of the change will be made, while if you enter 'S' for some, you will be given a choice at each instance to make the change or not by entering 'Y' for "Yes you want the change" and RETURN if you do not want to change. You can change single lines or line ranges by putting a single line number or range (e.g. 4,19) right after the C. Try it out with our little program in memory.

```
:C "JSR"JMP"
```

Press RETURN, and list the program. The JSR in line 1 is now JMP. To turn it back to JSR, from the Command Mode enter,

```
:C "JMP"JSR"
```

and press RETURN.

COPY. Sometimes you will write some code and want it repeated elsewhere in your program. Instead of having to key it in again, you can use the COPY function to do it automatically. Try the following with our two-liner in memory (If you have a 3-line program in memory, get rid of the first line with D1 {RETURN} using the D(elete) function.)

```
:COPY 1 TO 2
```

Press RETURN and list your source code. It will now look like this:

```
1 JSR $E544
2 JSR $E544
3 RTS
```

The COPY function placed Line 1 where Line 2 was and moved Line 2 to Line 3. You can also move code upwards. Let's put Line 3 in Line 1.

```
:COPY 3 TO 1
```

Now your listing looks like this:

```
1 RTS
2 JSR $E544
3 JSR $E544
4 RTS
```

You can also move a range of lines by specifying the range and the line where you want to insert the range. For example,

```
:COPY 10,20 TO 33
```

will duplicate the range of lines from 10-20 where Line 33 is. You probably won't be using this function a lot at first, but it sure saves a lot of time when you do need it.

MOVE. This function is just like COPY but it deletes the original line or range after it MOVEs it. This is handy when you find you have a line in the wrong order. To see how it works, using the D(elete) function, get rid of those lines in our program we added with the COPY function. (D1 {RETURN} and D2 {RETURN}). Now enter the following and press RETURN:

```
:MOVE 2 TO 1
```

When you list your program it now looks like this:

```
1 RTS
2 JSR $E544
```

Let's see if you can get it back to the original order using MOVE. (I'm not telling you how.)

R(eplace) The R(eplace) command is something like the D(elete) command except it puts you into the Insert Mode in the line you are replacing after first deleting the line. For example, enter the following and hit RETURN:

```
:R2
```

You will find yourself in Line 2. Hit the space bar and enter ABC as an opcode. (There's no such opcode is used simply for illustration.) When you list your program, it will look like this:

```
1       JSR  $E544

2       ABC
```

Now using the R(eplace) function, see if you can change the ABC back to RTS.

Finally, there are some miscellaneous other editing functions of Merlin you should understand. If you have a program in memory and you want to get rid of it, just enter NEW as in BASIC from the Command Mode. To toggle upper and lower case while in the Add/Insert mode, press the F7 function key. (You don't want lower case characters in the Command Mode.) Finally, to convert decimal numbers to hexadecimal numbers or hexadecimal numbers to decimal numbers from the Command Mode just enter the number you want converted and press RETURN. For example to convert hex to decimal, preface your number with a dollar sign ($). Let's say your program loads at $C000 and you want to know what the SYS value is. Do the following:

```
:$C000 {RETURN}
49152 = -16384
```

You can SYS either number 49152 or 16384 to run your machine language program after it's loaded into memory. To convert from decimal to hexadecimal try the following:

```
:1234
$04D2
```

You'll find this function very useful in converting back and forth between hexadecimal and decimal values.

LOADING AND RUNNING PROGRAMS

There are at least three ways to load and execute machine language programs created with Merlin. The easiest way is to LOAD the program from BASIC and then SYS the beginning address of your program. You have to use a special LOAD format, however.

```
LOAD "MACHINE PROG",8,1
```

Normally when you LOAD a BASIC program, you just enter,

```
LOAD "BASIC PROG",8
```

With machine programs, however, you have to add the ',1' after the '8'. The ',1' tells the program where to load in memory. The only problem with this method is remembering what the starting address of your program is. One trick I use when saving object code is to append the starting address to the end of the file name. So instead of just saving "MAC CODE", I'll name it "MAC CODE $C000" or "MAC CODE 49152" so that when I load it from BASIC I'll know what value to SYS.

Another way to execute object code is from Merlin. From the Executive Mode just choose 'G' for "Go!" and enter the object code file name when prompted with Run:. DO NOT put the '.o' extender on the name. Merlin does that automatically for you. The problem with this method is that you'll be popped back into the Executive Mode after the program executes. Also, if the code conflicts with the memory used by Merlin, you may crash!

Finally, you can load the source code, and then using ASM from the editor, assemble your code. Then just exit Merlin and SYS your program. This is a good method when you're still working on a program, since you can de-bug it by re-entering Merlin and working on the source code.

THE SOURCEROR

If you've been accumulating machine code programs on assemblers that do not save the source code, such as the Kids' Assembler or you have the object code but not the source code from a file, the Sourceror is going to be very valuable. It creates source code from object files, including labels!

To crank up the Sourceror enter,

```
LOAD "SOURCEROR.O",8,1
```

press RETURN and when the program is loaded enter,

```
SYS 49152
```

You will be prompted with the following:

```
Do you want an object file loaded?
(Y/N):
```

Take out your Merlin Master disk and put your work disk in the drive. Now press "Y" and when prompted, enter the name of your object file. Be sure to enter the full name, including any '.o' extender. Press RETURN and your program will be loaded. You will be told that start address and prompts what to do next. Make a note of the starting address of your program. Then you will be given a page of instructions of what to do. Enter the hexadecimal value of the beginning of your program and press the 'l' key. (That's the lower case 'L'.) For example, if your program begins at $C000, you would enter the following:

```
c000l
```

Press RETURN and your program will be disassembled. Find the end of your program, usually by locating an RTS instruction, and enter the hexadecimal address *below* the last instruction in your program and press 'q' for quit. For example, let's say your RTS is

at address $C0D0 and the next line, $C0D1, looks like garbage, you would key in,

```
c0d1q
```

and press RETURN.

The program will process your data and ask you for a file name. DO NOT add the '.s' extender to your file name since Sourceror does that automatically. You have just saved the source code for your object code! Now you can load and run Merlin and look at the source code in your editor.

Once you load your Sourceror created source file in Merlin, list it. You will notice that some of the hexadecimal numbers are prefaced by an "H". This simply means they are (H)exadecimal numbers. Using your editor, replace the H's with a dollar sign and you're all ready to assemble it into a clear source file. You can add labels and comments if you want as well. The following shows what a source file might look like and how to change it.

```
1        ORG     $C000
2
3        JSR     HE544
4        STA     HD021
5        RTS
```

In lines 3 and 4, change the HE544 and HD021 to $E544 and $D021. When you're finished, your code should look like the following:

```
1        ORG     $C000
2
3        JSR     $E544
4        STA     $D021
5        RTS
```

Also, you might have some garbage at the end of your file. Just use the (D)elete function from the command mode to get rid of it. ASM your code and save it as a new source file.

MERLIN'S MONITOR

To test run your assembled programs, the monitor in Merlin is quite handy. To enter the monitor from the Command Mode, enter,

```
:MON
```

and press RETURN. You will be given a '$' prompt to indicate you're in the monitor. All values are expected to be given in hexadecimal. (The prompt will remind you of that.)

The best way to use your monitor as a test bench is to leave out the ORG directive in any program you're testing. In this way, the default ORG of $8000 will be used and the monitor, editor, assembler and your code can be co-resident in memory. Once you have ASseMbled your source code, enter the monitor, key in,

```
$8000g
```

and press RETURN. (Remember the dollar sign ($) is the prompt; so you don't have to include it.) Your program will now execute. Once your program is debugged you can use any ORG you want by inserting it in your source code and re-ASseMbling it.

You can do a lot more with the Merlin monitor, but our main purpose is to use it as a test bench. Take a look at your Merlin manual for its other uses. To get back into your Editor, enter 'r' and press RETURN. If you enter 'q' {RETURN} you'll be returned to the Executive Mode.

SOME EXAMPLES

To get you rolling, let's look at a couple simple examples.The first one just clears your screen and changes the printing characters from blue to black. Actually, it works about the same as using CHR$ codes to put things on your screen.

```
1          ORG     $C000
2          JSR     $E544     ;Jumps to clear
                             routine
3          LDA     #144
4          JSR     $E716     ;Jumps to screen
                             output
5          LDA     #65
6          JSR     $E716
7          LDA     #66
8          JSR     $E716
9          RTS               ;Returns from
                             subroutine to BASIC

:ASM
```

When you are finished assembling the program, enter 'Q' to return to the Executive Mode and then save both the source file and object file. When you load and execute the program it will print black letters, AB in the upper left hand corner of your screen.

This next program changes your background color to black, your border to purple and prints white letters. There's a trick involved. Notice that Line 2 is blank. To get blank lines with Merlin, you have to hit the space bar once when you come to a new line and then RETURN. (If you just hit RETURN without the space bar, you'll go into the Command Mode.)

```
1          ORG     $C000
2
3 START    JSR     $E544
4          LDA     #0        ;Color code for black
5          STA     $D021     ;Store in background
                             reg.
6          LDA     #4        ;Color code for
                             purple
7          STA     $D020     ;Store in border reg.
8          LDA     #5        ;ASCII control code
                             for white letters
9          JSR     $E716     ;Output to screen
10 END     RTS
:ASM
```

You're not expected to know how this works, but you soon will. The START and END labels really don't do anything in this example, but later we'll see how easy they can make life in assembly language programming.

CHAPTER 4

THE COMMODORE 64 MACRO ASSEMBLER DEVELOPMENT SYSTEM

THE PARTS

To understand the Commodore assembler package, the first thing to understand is that it is in several parts. When you load one part, you do not necessarily have access to another. That is, you have to separately load the various files that make up the package. The major elements of the package include the following:

1. EDITOR64
2. ASSEMBLER64
3. CROSSREF64
4. LOLOADER64
5. HILOADER64
6. MONITOR$8000
7. MONITOR$C000

The disk version also includes the DOS WEDGE64, a disk operating system. The most relevant items are EDITOR64, ASSEMBLER64 and the two loader programs. We'll discuss their use in detail with only secondary attention to the other files. Since editing code works a lot like editing a BASIC program, it's a good idea to load the DOS WEDGE64 and use it just as you would when

programming in BASIC. The easiest way to do that is to LOAD "BOOT ALL",8 and then RUN it.

EDITOR64

You write your assembly language programs in the EDITOR64 mode, save the source code to disk and then assemble the code with ASSEMBLE64. Since these two operations require two different files, we will treat their operation as two distinct events. To start, enter the following:

```
LOAD "EDITOR64",8,1 {RETURN}
SYS 49152
```

After you do that, your screen should look like this:

```
   COMMODORE 64 EDITOR V07282
(C) 1982 BY COMMODORE BUSINESS MACHINES
READY.
```

It looks like nothing has happened, but you are all set to start writing assembly language code. You do this by entering line numbers just as in BASIC. The four fields are:

1. Label
2. Opcode
3. Operand
4. Comment

They are arranged as follows:

```
LABEL    OPCODE    OPERAND    ;COMMENT
```

The COMMENT field is indicated with a semi-colon before the comment. Each time you want to go to a different field, you simply press the space bar. We will use line numbers beginning with 10 and incremented by 10. This will give us room to insert code if we need it. To get started enter the following:

```
AUTO 10 {RETURN}
```

Nothing will appear to happen at this point. Now, just start writing code beginning with line 10.

```
10 JSR $E544
```

Put two spaces after the line number and one space after JSR. As soon as your press RETURN, your screen should look like this:

```
10 JSR $E544
  20
```

The new line number, 20, automatically pops up. You are in the LABEL field; so press the space bar to get into the OPCODE field and enter the following:

```
10 JSR $E544
  20 RTS
```

You have just written a complete (but small) assembly language program. The first line jumps to a built-in subroutine to clear the screen and the second line returns you to BASIC. However, the Commodore assembler requires a special pseudo-opcode at the end of all programs, .END. So press RETURN and enter the following:

```
10 JSR $E544
  20 RTS
  30 .END
```

Now when your code is assembled with ASSEMBLER64, it will know the end of the program. However, it also needs the beginning of the program. Since the beginning should be at the beginning, we will insert a line just as we can do in BASIC. So enter the following line:

```
5 * = $C000
  15 ◄Just press RETURN when you get the 15}
```

Now LIST your program and it should read:

```
5  * = $C000
10 JSR $E544
20 RTS
30 .END
READY.
```

The asterisk (*) is the symbol used to define the starting address of your program. (Many assemblers use ORG as a psudeo-opcode instead of the asterisk.) The dollar sign ($) in front of the address indicates it is a hexadecimal value. You could have entered,

```
5  * = 49152
```

if you wanted a decimal value.

Next, since this still doesn't look very much like an assembly language program, let's format it. Just enter,

```
FORMAT
```

and press RETURN. Your program now appears as this:

```
5       * = $C000
10      JSR $E544
20      RTS
30      .END
READY.
```

As you write your code, it is a good idea to FORMAT it every now and then so that you can better see the separate fields. We have not put anything in the LABEL field yet, but if we did, the spaces between the line numbers and OPCODEs would have our labels. Similarly, if you add a comment, it too will be formatted to the correct field.

Since we inserted our starting address for the code, our line numbers are out of whack. We can reNUMBER them with the NUMBER command. Do the following:

```
NUMBER5,10,10
{RETURN}
```

After NUMBER the first number is the old start line, the second number is the new start line and the last number is the step size. So, since our old first number was 5, and we wanted our new first number to be 10 and we wanted our line numbers incremented by steps of 10, we used the 5,10,10 combination. Just remember the format for NUMBER as:

```
NUMBER(Old start),
(New start),(Increment)
```

I'd like to renumber please.

Now when you LIST your program it looks like this:

```
10 * = $C000
20 JSR $E544
30 RTS
40 .END
READY.
```

= = MAKING A HEADER = =

While you're learning assembly language, it might be a good idea for you to make a header that describes the different fields for you. You could put in Line 1 something like the following:

```
1 LABEL OPCODE OPERAND ;COMMENT
```

When you FORMAT your code, these fields will line up over your code to show you if you have your labels, opcodes, operands and comments in the correct places. Before you save your program to disk, just delete Line 1 so it will not be assembled and mess up your object code.

Now your program is all set to assemble into object code. (Object code is the machine language code that will run.) To do this you have to first put your source code (the code you just wrote) onto disk. Rather than using the SAVE command, you use PUT or CPUT. So let's try it.

```
PUT "TEST1.S"
```

You can optionally include parameters for starting line, ending line, device number (the default is 8, your disk drive) and the secondary address. If you use CPUT instead of PUT, your file will be more compact on your disk. (Think of it as CompactPUT.)

At this point you can assemble your program. You are finished with everything you have to do in the EDITOR64. However, let's look at the other editing features before we go on.

For the most part, you edit assembly source code just as you would a BASIC program. Thus, there are no special editing commands for normal source code editing. However, there are some very useful added editing functions in EDITOR64. Let's look at them.

= = DOS WEDGE64 HELPS A LOT = =

It's a very good idea to load your DOS WEDGE64 when you're working in the editor. With it, you can look at your directory without destroying your program in memory. Thus when PUTting a file to disk, you can first look at the disk directory with > $ to see what file names are taken up already. When you GET a file from disk, you can check to see if the file name you want is on the disk.

Look at your directory to make sure your source code is saved as a SEQ file and then enter NEW to clear memory. Now, using the GET statement, we'll see how to load source code from your disk. Enter,

```
GET"TEST1.S" {RETURN}
```

Your program will be loaded to disk, and you can LIST it. When you do, you'll see the following:

```
1000 * = $C000
1010 JSR $E544
1020 RTS
1030 .END
READY.
```

As you can see, your lines now start at 1000 instead of 10. Your editor automatically does this for you. (Don't ask me why.) Anyway, when you're working on source code, you can save a part of it and then re-load it with GET and continue your work.

ADDED EDITING FUNCTIONS ON EDITOR64

CHANGE. Sometimes you will want to make a lot of changes in your program. To do this quickly with a single string, the CHANGE command is very helpful. Let's say you accidentally used LAD instead of LDA. The CHANGE command would go through your program and change all instances of LAD to LDA. Using your program in memory, we'll change JSR to JMP. Do the following:

```
CHANGE/JSR/JMP/ {RETURN}
1010 JMP $E544
```

The changed line pops up showing you that your JSR is now JMP. With just one change, it's probably just as easy to edit as you would in BASIC. However, when you have several opcodes, labels or other strings to change, you will really appreciate this function.

DELETE. This command would be handy with BASIC programs. If you want to get rid of a range of lines, you just enter, for example,

```
DELETE 1000-1030 {RETURN}
```

and lines 1010-1030 will be deleted. This is very helpful when you find a way to tighten up your code and you want to get rid of extra lines. All you have to do is enter the first line to the last line of code you want DELETEd.

FIND. When your programs get really long and you want to find a string, the FIND command is a big help. (If you DELETEd your lines in the above example, you better GET "TEST1.S" back into memory.) Enter, for instance,

```
FIND "RTS" {RETURN}
1020 RTS
```

Notice that the FIND command requires quotation marks around the string you are trying to find, while the CHANGE command does not want the quotation marks.

Finally, if you want your EDITOR64 out of action, just enter KILL {RETURN}. You can get it back with SYS 49152. That's about it for editing. Remember, you edit most of your code just as you would a BASIC program. The added editing commands simply make it a lot easier for you.

ASSEMBLER64

Now that you have your source code saved as TEST1.S, you are ready to assemble it with ASSEMBLE64. To get started do the following:

```
LOAD "ASSEMBLER64",8 {RETURN}
RUN {RETURN}
```

Unlike the EDITOR64, you do not have to SYS an address to get it up. You just RUN it. As soon as you RUN the program you will see,

```
CBM RESIDENT ASSEMBLER V080282
(C) 1982 BY COMMODORE BUSINESS MACHINES
OBJECT FILE (CR OR D:NAME)
```

Now BE CAREFUL here! You are asked to enter the name for the OBJECT file. You saved the SOURCE file to disk under the name TEST1.S, and you DO NOT want the same name for your OBJECT file. Enter the name TEST1.O and hit RETURN. By using the extenders '.S' for source files and '.O' for object files, you won't get into as much trouble.

Next you will be asked,

```
HARD COPY (CR/Y OR N)?
```

If you have your printer hooked up, enter Y or just press RETURN. If you do not have a printer hooked up and on-line enter N and press RETURN. Next you will be asked,

```
CROSS REFERENCE (CR/NO OR Y)?
```

You won't need a cross reference until you get into more complicated programs; so enter N or just RETURN.

Now you will be asked the SOURCE file name.

```
SOURCE FILE NAME?
```

You saved your source file as TEST1.S so enter that name and press RETURN. Your screen will show the following:

```
PASS1
TEST1.S.....PAGE 0001
LINE# LOC CODE LINE
PASS2
```

```
00001           0000                    * = $C000
00002           C000        20 44 E5    JSR $E54
4
00003           C003        60          RTS
00004                                   .END
ERRORS = 00000
END OF ASSEMBLY
READY.
```

If that's what you got, you correctly entered and saved your source code, assembled your object code and saved the object code to disk in a SEQ file. Look at your directory to make sure. You should see both "TEST1.S" and "TEST1.O". If you did not get that, then enter SYS 49152 to get back into the editor, GET your source code, make corrections and try it again.

LOADERS

One thing I'm not crazy about with this assembler is the way object code is loaded and executed. Most object code can be loaded with LOAD "FILENAME",8,1 and then SYS the load address. However, since the files are stored as SEQ files instead of PRG files, that's not possible. (It *is* nice for cassette users though.) You have to load either LOLOADER64 or HILOADER64. If you load your program in higher memory locations, such as we did at $C000 (49152), use the LOLOADER64 since it will place itself in a position that will not conflict with the code up in $C000. (It loads at $800). If your code is in a

lower address, such as the cassette buffer at 828, then use the HILOADER64. To crank them up you have to use two different formats:

```
LOAD "LOLOADER64",8,1 {RETURN}
RUN {RETURN}
LOAD "HIGHLOADER64",8,1
SYS 51200
```

Let's do it and see if our program works. Load the LOADER64 and RUN it. Your screen will look like this:

```
LOLOAD.C64 V072772
(C) 1982 BY COMMODORE BUSINESS MACHINES
HEX OFFSET (CR IF NONE)? {RETURN}
OBJECT FILE NAME? TEST1.O {RETURN}
C000
C003
END OF LOAD
READY.
```

The HEX OFFSET refers to the number you want to add to your load address. If you wanted to load your program at $C200 and it was already at $C000, your HEX OFFSET would be $200. Just press RETURN since we want it to load at $C000 (49152). Enter the filename, TEST1.O, press RETURN and you will see the beginning address of your program and the end of it (C000-C003). Now we're all set to give it a go. Hold your breath and enter,

```
SYS 49152
```

If your screen clears and all you can now see is the READY. prompt, your program works. If it locks up your screen, blows out the windows and sets your hair on fire, try again.

THE MONITORS

Like your loaders, the Commodore assembly language package comes with two monitors. MONITOR$8000 loads in the low por-

tion of RAM and MONITOR$CØØØ in the high portion. Load the monitor in the low portion of memory and we will use the high RAM for some examples.

First of all, you may be wondering what a monitor is. Basically, a monitor is a device that allows you to examine and change your code in memory, usually in hexadecimal values. Monitors also enable you to disassemble machine language programs in memory and see the status of the various registers in your microprocessor. (We'll discuss registers in Chapter 6.)

Your Commodore monitors can also be used as assemblers, AND they can save machine code as PRG files. This means that you can load these files directly from BASIC without having to use the loader programs. (Why didn't they do this with ASSEMBLER64?!) Therefore, if you want to assemble your code so that it can be stored as a PRG file, you can use the mini-assembler in your monitor to write assembly programs.

First, LOAD "MONITOR$8ØØØ",8,1 and SYS 32768. When you first get into your monitor, you will see the following:

```
8*
  PC   SR   AC   XR   YR   SP
.;803E 32   ØØ   83   84   F6
.
```

The period (.) is your monitor prompt, awaiting your command. The PC, SR, etc. at the top refer to:

1. PC = Program counter
2. SR = Status register
3. AC = Accumulator
4. XR = X register
5. YR = Y register
6. SP = Stack pointer

The values of the various registers is the number immediately below the initials. For example, the accumulator has 'ØØ' in it. When you

get to Chapter 6, you can use your monitor to examine these registers.

For now, let's see how to assemble code in the monitor. The command to enter assembly code is 'A'. Do the following to get started:

```
.A C000 JSR $E544 {RETURN}
```

As soon as you press RETURN your code is assembled and your line now looks as follows:

```
.A C000 20 44 55 JSR $E544
.A C003
```

The numbers to the left of your mnemonic opcode and operand are the machine language opcode and operand. The next available address is C003, and so you're all set to enter more assembly code. FIRST hit the space bar and then enter the next line. After each RETURN, you have to hit the space bar first before entering your opcode.

```
.A C000 20 44 55 JSR $E544
.A C003 LDA #$00 {RETURN}
.A C005 STA $D021 {RETURN}
.A C008 RTS {RETURN}
```

Each time you hit return, the line you had just entered will insert the machine language code. When you're finished with the above program will look like this:

```
.A C000 20 44 55 JSR $E544
.A C003 A9 00    LDA #$00
.A C005 8D 21 D0 STA $D021
.A C008 60       RTS
.A C009
```

Compared to using EDITOR64, this is a little messy, and it assembles your code each time you hit return. You can edit your code, but inserting code between addresses is complicated. For the

time being, let's just see how to save a PRG file using the above program.

To save a machine code program as a PRG file, simply press RETURN after you are finished to get the period (.) prompt without the 'A' and address. The 'S' command is used to S(ave) PRG programs to disk or tape. Use the following to save the above example:

```
.S"MONPRG.O",Ø8,CØØØ,CØØ9 {DISK}
.S"MONPRG.O",Ø1,CØØØ,CØØ9 {TAPE}
```

The parameters after the file name include:

1. Device number; Ø1 for tape and Ø8 for disk.
2. Starting address.
3. Ending address *plus* 1.

Notice in our little program that we started at $CØØØ and ended at $CØØ8. Since $CØØ8 + 1 = $CØØ9, $CØØ9 was our value used to Save the program. Until you become comfortable with adding in hexadecimal, just use the last address on the screen before you left the 'A' mode.

To test your program from the monitor, you use the 'G' for "Go" command. Since our program began at $CØØØ, we would enter,

```
.G CØØØ {RETURN}
```

You should have seen your screen clear and turn black. The little program simply loaded the value for black (Ø) into the address for the background color ($DØ21). To get back to the monitor, just key in SYS 32768.

To see if your program really was saved as a PRG file, turn off your computer and disk drive and re-start it. Load your program as follows:

```
LOAD "MONPRG.O",8,1
```

When you get your READY. prompt, key in:

SYS 49152 {RETURN}

If your program is correctly saved, it should immediately clear your screen and turn it black. That's a lot easier than using the LOLOADER or HILOADER programs to get your machine program in memory.

= = CONVERTING YOUR EDITOR64 FILES = =

If you want to convert your SEQ object code files created with EDITOR64 and ASSEMBLER64 into PRG files, you can do so (painstakingly) with the monitor. First, load your SEQ object program with LOLOADER or HILOADER depending on where your files loads. Then, load the corresponding monitor program. (MONITOR$8000 if LOLOADER and MONITOR$C000 if HILOADER.) Disassemble your object code using the D command and starting and ending address of your object file. For example, you would enter the following for our program:

.D C000,C008 {RETURN}

If you're not sure of the ending address, just disassemble your code a few addresses at a time until you find the end. Then using the S(ave) command from your monitor, save your program using '.OP' (for OBJECT PRG) as an extender.

Other Monitor Commands

Beside the A, G and S commands, your monitors have several others as well. Here's a summary of them.

L for L(oad). This re-loads PRG files created with monitor Save. Use the following format:

.L"FILENAME",DEVICE NUMBER

The device number is usually Ø8 for disk and Ø1 for tape.

.D for D(isassemble). This disassembles code from a specified starting address to a specified ending address. For example, we could look at our illustration program with the following:

.D CØØØ CØØ8 {RETURN}

.M for M(emory dump). This is something like disassemble, but you are given the hexadecimal machine code in memory locations specified. For example, dump our example program to memory with the following:

.M CØØØ CØØ8 {RETURN}

Compare it with what happened when you used the D command.

.R for R(egister Display). When you learn about registers in Chapter 6, you may want to see the status of the various registers. You just enter R and press RETURN.

There are some other monitor commands we won't get into since they are used for larger programs than we will deal with in this book. Practice with the ones we've discussed to learn how machine code is stored in memory.

SOME EXAMPLES

O.K., crank up your EDITOR64 and let's see some examples. We'll use the comment and label fields. To get going enter AUTO 1Ø {RETURN}.

```
1Ø    ;LABEL  OP   OR     COMMENT
2Ø    * = $CØØØ  ;LOAD  ADRS
3Ø    JSR  $E544
4Ø    LDA  #Ø
5Ø    STA  $DØ21
6Ø    LDA  #5
7Ø    STA  $E716
8Ø    RTS
9Ø    .END
```

Our entire first line is a COMMENT. The semi-colon lets the assembler know that it is only source code that is not to be compiled. When you FORMAT your code, you can see if your assembly code lines up under the correct label. If it does not, edit the lines after LISTing them. Also while editing, turn off AUTO by just entering AUTO {RETURN}. The FORMATted listing should appear as the following:

```
10    ;LABEL  OP   OR          COMMENT
20            * = $C000        ;LOAD ADRS
30            JSR $E544
40            LDA #0
50            STA $D021
60            LDA #5
70            STA $E716
80            RTS
90            .END
```

We did not use the LABEL field, but you can see the spaces where labels would be when FORMATted. By frequently FORMATting your code, you can check for the correct fields. Now PUT this on disk and assemble it, and then load and run it. It'll turn your screen black and your letters white. The next one turns your border purple and your background yellow. (It looks like your sister's first attempt applying make-up.)

```
10    ;LABEL   OP   OR        COMMENT
20    * = $C000 ;LOAD ADRS
30    JSR $E544
40    LDA #4 ;PURPLE
50    STA $D020 ;BORDER ADRS
60    LDA #7 ;YELLOW
70    STA $D021 ;BACKGROUND ADRS
80    RTS
90    .END
```

With more comments, you have a better idea of what's going on in the program. Later when we need labels, we'll see how they make loops very easy. Now, go on to the next chapter and take a look inside your computer.

CHAPTER 5

STRANGE NEW NUMBERS

DECIMAL, BINARY AND HEXADECIMAL NUMBERS

If you've been following our examples, you probably noticed some funny numbers with dollar signs in front of them, like $E544 and $D021. Now with a perfectly good numbering system we all understand, why confuse matters with funny numbers called hexadecimals? The reason lies in some even weirder numbers called "binary" and the workings of your microprocessor.

To begin with, suppose instead of counting on our 10 fingers, we used just our two arms. After all, the 10 digits in our numbering system is based on our 10 fingers. Since we're used to our 10 fingers, our numbering system seems natural. However, our two arms are natural, and we could have just had to remember two digits instead of ten if we had counted with our arms instead of our fingers. Let's count from 0 to 10 with our decimal numbering system and think about what's going on:

```
0
1
2
3
```

4
5
6
7
8
9
10

Everything is lined up fine until we get to 10, and then we have to start all over again with our 10 digits (0-9) adding a zero onto our second digit. To keep things neat, let's add a zero before our first digit:

00
01
02
03
04
05
06
07
08
09
10

That's much better since our numbers are all lined up. Now all we did in this system is clear. With our 10 digits (0-9), we use all of them in our far right column, and when we run out, we simply tack on a 0(start over) and increment our second from right digit by 1. How would the same thing look with only two digits, 0 and 1?

00
01
10
11

We ran out of digits pretty quick! Compare that with our decimal system:

00	00
01	01
10	02
11	03

In other words, using our arms instead of our fingers, we can only count to 3 before we run out of digits and have to add a third digit.

Okay, so we can count by a two digit system instead of the ten digit system. But why? The reason lies in the way your computer really works. Basically it reads electrical currents as either being on or off. If a current is read as "ON" then it is read as "1." If it is read as "OFF" it is read as "0." Now if you think computers are pretty dumb, you're right. However, while they may be dumb, they're very fast. As a result, your Commodore 64 can interpret a bunch of 0's and 1's into decimal numbers and strings and look like it's really smart.

Your 6510 microprocessor is an 8 bit device. That means that it has 8 cells that read a current in each cell (bit) as being ON or OFF. Just about everything else in the computer is arranged to deal with those 8 bits in the microprocessor. Knowing this, we know that we can have a two digit numbering system 8 digits long.

```
7 6 5 4 3 2 1 0
0 1 0 1 0 1 0 1
- + - + - + - +
```

The plus and minus signs underneath the values indicate whether the current is ON or OFF. The 8 values (7-0) above the digits represent the way byte is arranged. Thus when we say that Bit 6 is ON, we know we're talking about the second bit from the left.

Our two digit binary system can count from 00000000 to 11111111. In decimal, that's 0 to 255. However, since 0 to 255 is an uneven system, computer programmers used a 16 digit system

numbered from $Ø to $FF. To see why, let's break down our 8 bit number into two parts. Each part is called a "nibble".

```
7654    3210
0101    0101
```

Now, let's see how much we can pack into a two digit system with four places and put our decimal and hexadecimal numbers next to them. Binary numbers are often indicated with a percent sign; so let's include one here.

Binary	Decimal	Hexadecimal
%0000	0000	$0000
%0001	0001	$0001
%0010	0002	$0002
%0011	0003	$0003
%0100	0004	$0004
%0101	0005	$0005
%0110	0006	$0006
%0111	0007	$0007
%1000	0008	$0008
%1001	0009	$0009
%1010	0010 ◄-	$000A
%1011	0011	$000B
%1100	0012	$000C
%1101	0013	$000D
%1110	0014	$000E
%1111	0015	$000F

Now let's look again, but this time, we'll clear all the leading zeros from the hexadecimal values and leave out the decimal.

Binary	Hexadecimal
%0000	$0
%0001	$1
%0010	$2
%0011	$3
%0100	$4
%0101	$5

%0110	$6
%0111	$7
%1000	$8
%1001	$9
%1010	$A
%1011	$B
%1100	$C
%1101	$D
%1110	$E
%1111	$F

Sweet $10

As you can better see, the hexadecimal numbering system can represent an entire nibble with a single digit. The decimal system has to start using 2 digits when the binary system counts to 10 10.

```
1111 ◄-Binary
   F ◄-Hexadecimal
```

Now since an 8 bit grouping, which is called a "byte", is composed of two nibbles, we'll have to see what maximum number in hexadecimal can be stored in a byte.

```
1111 1111 = 11111111
  $F   $F = $FF
255 x 256 + 255 = 65535
```

Since there are over 64000 bytes in your Commodore 64, a number system that keeps track of what is in each byte with only two digits is more efficient than one that takes up to five digits, as would the case be with a decimal system. (By the way in case you didn't know, the "64" in Commodore 64 stands for 64 kilobytes. A kilobyte is 1024 bytes. Thus your Commodore has 64 x 1024 bytes of RAM.)

At this point, no one expects you to understand everything about these weird numbering systems. Rather, all you should be able to grasp is the concept that its easier to use a maximum of two digits to represent what is in a byte than five. There are all sorts of conversion programs that will allow you to convert numbers into good old decimal numbers. Your assembler will do it automatically for you.

Later on when you've worked with hexadecimal and decimal numbers for a while, it will be a lot simpler. In the meantime, you can use decimal numbers in assembly programs if you want.

To re-cap, here's what's going on in your computer and why hexadecimal numbers are easier to use:

1. The processor interprets electrical currents as being ON or OFF
2. The ON/OFF configuration is stored in an 8 bit binary number or byte.
3. The 8 digit binary number is represented as a two digit hexadecimal number.
4. When you look at a disassembly of machine programs, you see hexadecimal numbers representing values in bytes.

GOING BETWEEN NUMBER SYSTEMS

Using conversion charts

Probably the best way to convert numbers when you're working with your computer is with number conversion charts. In this way, you can keep your computer free for programming and still make the conversion. In the Appendices, there are several charts which give addresses in both decimal and hexadecimal. In this way, you can quickly find an address, the token of a machine language opcode or other information in both decimal and hexadecimal. However, since the computer works with bytes, and each byte can only hold a value of $FF or 255 (actually 256 since you can enter values from Ø-255), you have to make conversions on your own when the number is greater than 255. This will help you understand something about how the computer "thinks," and how you have to think in writing assembly language programming.

Now since all the computer can stuff in a single byte is 255, how does your computer deal with numbers larger than 255? It has to use two bytes. One is called the high byte and the other is called the low byte. Let's look to see how 256 looks in a two byte, high byte-low byte configuration:

High Byte	Low Byte	
76543210	76543210	
00000001	00000000	◄-Binary
$01	$00	◄-Hex
1	00	◄-Decimal

By multiplying the decimal number in the high byte by 256, we can get the high byte value of our decimal number. Since the Low byte is zero in our example, we just add the high byte and low byte to get 256. The hexadecimal value doesn't have to be changed at all. We just tack on the two zeros in the low byte to the $1 in the high byte to get $100, the hexadecimal value of decimal 256. Now suppose in PEEKing at a couple of addresses we found a high byte-low byte value of the following:

High Byte	Low Byte
12	33

To determine what that is in hex is actually easier than determining the decimal value of the combined number. Looking at a decimal-hex conversion chart, we find that the 12 is $0C and the 33 is $21. The hex value is $0C21. The decimal value is 256 x high byte + low byte. Thus, 12 x 256 + 33 = 3105, and now we know that $0C21 = 3105. We did that with only a conversion chart from 0-255 ($0-$FF).

Now, how about going the other way? Let's say we have the hex number $ABCD. First, we divide it into high byte and low byte getting:

High Byte	Low Byte
$AB	$CD

Our conversion chart shows us that $AB = 171 and $CD = 205. Now we have a decimal high byte of 171 and low byte of 205. We multiply 171 x 256 and then add 205 to get 43981. Your assembler takes care of these matters for you, but as you get further and further into machine and assembly language programming, you'll be looking up more and more numbers.

CONVERSION PROGRAMS

Sometimes you will be planning out a program and want to look up values quickly. You won't be entering code on your computer, but instead planning a program on paper making a lot of conversions between hex and decimal. While the charts are handy, you want something a little simpler. (This is especially true if you want to POKE or PEEK values in BASIC.) The following program will do that for you. Pay special attention to the algorithms beginning in Lines 240 and 300.

HEX/DEC DEC/HEX CONVERSION

```
10 PRINT CHR$(147)
20 INPUT "DEC OR HEX CONVERSION (D/H)Q = QUIT
   ";C$
30 IF C$ = "H" THEN 200
40 IF C$ = "Q" THEN END
50 IF C$ < > "D" THEN 20
60 HEX$ = "" : N = 0
70 INPUT "DECIMAL VALUE";N
80 HB = INT(N/256)
90 LB = N-INT(N/256)*256
100 FOR X = 1 TO 2
110 IF X = 1 THEN N = HB
120 IF X = 2 THEN N = LB
130 N% = INT(N/16) :GOSUB 310
140 N% = N-N%*16:GOSUB 310
150 IF X = 1 THEN H1$ = HEX$ : HEX$ = ""
160 IF H1$ = "0" THEN H1$ = "00"
170 NEXT
180 HEX$ = H1$ + HEX$
190 HEX$ = "$" + HEX$ :PRINT "HEX = "; HEX$:
PRINT: GOTO 20
200 H$ = "" : DE = 0
210 INPUT"HEX VALUE ";H$
220 GOSUB 250
230 PRINT "DECIMAL VALUE = ";DE
240 PRINT : GOTO 20
250 REM **********************
```

```
260 REM CONVERT HEX TO DECIMAL
270 REM **********************
280 FOR L=1 TO LEN(H$) : HD=ASC(MID$(H$,L,1))
290 DE=DE*16+HD-48+((HD>57)*7)
300 NEXT L : RETURN
310 REM **********************
320 REM CONVERT DECIMAL TO HEX
330 REM **********************
340 HEX$=HEX$+CHR$(48+N%+7*ABS(N%>9))
350 RETURN
```

When you RUN the program, just choose 'H' to convert from hexdecimal and 'D' to convert from Decimal-hex.

Binary-Hex-Decimal

In converting between hex and decimal, we broke down the hex value into the high byte and low byte values. With binary conversions, we'll break 8 digit binary values into 4 bit nibbles. Each nibble converts into a single digit hex value. Then, by putting the hex values in each nibble together, we have a one byte hex value.

Byte

High Nibble	Low Nibble
7 6 5 4	3 2 1 0
0 1 0 1	1 1 1 0

Going back in this chapter to our binary-hexadecimal comparison, we see that the high nibble's binary value of 0101 (or %0101 since

High Nibble

Low Nibble

binary values are indicated with a percent sign) is $5 and the low nibble value of %1110 is $E. Thus, the final hex value is $5E. You can translate this into decimal by using the hex-decimal conversion.

If you don't already know how to convert directly from binary to decimal, here's a simple way to do so. Take the byte as a unit and use the multiples above the specified bit. After finding the multiple value of each bit, add them all up to get your decimal value.

```
128   64   32   16   8   4   2   1   <- Multiple
-----------------------------------------------
 7    6    5    4    3   2   1   0   <- Bit
-----------------------------------------------
 0    1    0    1    0   1   0   1   <- Binary
                                       number
-----------------------------------------------
0 + 64 + 0 + 4 + 0 + 4 + 0 + 1 = 73 Decimal
```

If you understand powers the bits can be understood as follows:

```
2^7  2^6  2^5  2^4  2^3  2^2  2^1  2^0  <- Power of
                                          two
 7    6    5    4    3    2    1    0   <- Bit
```

Notice how the power of two corresponds with the bit number. That way you can remember the multiple simply by knowing the bit number. Thus instead of remembering that bit 5 is 32, you can remember it as 2^5 or "two to the fifth power." (On your Commodore 64 the ^ is represented by the up-pointing arrow.) Knowing that, writing a little conversion program in BASIC should be a snap:

```
10 PRINT CHR$(147)
20 INPUT "BINARY VALUE "; B$
30 IF LEN(B$) <> 8 THEN 20
40 FOR X = 0 TO 7
50 V$ = MID$(B$, X+1, 1)
60 V=VAL(V$) : IF V < 1 THEN X=7: PRINT
"BONG!":TD=0:BV=0:NEXT:GOTO 20
```

```
70 P = 7 - X
80 IF V = 1 THEN BV=2↑P : REM UPWARD ARROW
KEY
90 TD = TD + BV
100 BV=0
110 NEXT X
120 PRINT "DECIMAL VALUE="; TD
130 INPUT "ANOTHER(Y/N) ";AN$
140 IF AN$ ="Y" THEN TD=0 : GOTO 20
```

(If you work with sprites, the above program will help you find their values quickly.)

HOW NOT TO WORRY ABOUT NUMBERS

Your assembler will accept just about any kind of numbers; so, if you know the correct number, you can enter it in either hex or decimal. As we go along we'll be discussing more and more hexadecimal elements of programming in assembly language because it is easier than decimal. (If you don't think it's easier, just look at a batch of numbers with binary patterns. With fewer digits, the numbers get longer quicker!) At this point, though, do not be overly concerned. Why worry if you can enter your programs in either format. (Merlin and the Commodore assembler even accept binary numbers.)

This chapter has just been a quick introduction to the number systems used in computers. As you need it, you can always come back and look up what you want. Don't try and attempt to be adept at converting numbers all at once. Using your charts and programs, you will gradually be able to see how to convert decimal into hexadecimal and vice versa. In the meantime, just forge ahead and enjoy your computer.

CHAPTER 6

WHAT'S IN YOUR MICROPROCESSOR?

The 651Ø contains seven registers, three of which we will be using a great deal and four others which you should begin to understand. We will be manipulating the A, X, and Y registers in our programs a lot, while the other registers will affect what we're doing to a more or less apparent degree. But to get started, let's begin with the notion of a register.

WHAT'S A REGISTER?

Essentially, a register is something that keeps track of something else. For example, stores keep track of the money they receive with a cash register. It records how much money went in and how much went out as change or refunds. Your major registers in your microcomputer are much simpler in that most of them can only count up to 255 ($FF) before they run out of room. In the last chapter, we saw that your computer operates in 8 bit chunks called "bytes." Most of the registers in your microcomputer also work in 8 bit bytes.

Since the registers can handle so little, you have to tell them everything you want them to do. Think of them as some not-too-bright friend who must be told everything. For example, let's say you ask your little brother, "Go get my baseball bat." That's all you need to say since you assume that he knows where the bat is. In BASIC you can do pretty much the same thing with statements such as GOTO, GOSUB and IF/THEN. BASIC and other high level languages are really smart compared to assembly language. With assembly language, you have to tell it *everything* you want. If your little brother was given instructions in assembly language, "Go get the bat," would become something like the following:

1. "Go into the house and up the stairs."
2. "At the top of the stairs, turn left and go into my room."
3. "In the room, go to the right wall where you will find a rack."
4. "On the rack is a baseball bat"
5. "Put the bat in your hand and take it out of the rack."
6. "Turn around and leave the room."
7. "Come down the stairs and out of the house."
8. "Come to me and hand me the bat."

The reason you had to give this large set of instructions is because your little brother can only handle one byte at a time. (Once you give him the instructions, though, the little kid sure is fast!)

The instructions in assembly language are similar to giving instructions to someone who can only handle one small parcel at a time. The size and complexity of the instruction is dependent on the size of the register. Since most of the registers are eight bits, the instructions must be relatively small.

THE ACCUMULATOR

The most important register is the accumulator or A register. Its contents are read by other registers, and it is used as a cross-roads in your machine. It performs addition and subtraction, logical operations and is usually the register to send information to different memory locations. Several mnemonic opcodes refer to the Ac-

cumulator. For example, the instruction LDA LoaDs the Accumulator (or A register) with some value, and STA STores what's in the Accumulator at some address. Many instructions check to see what's in the accumulator before taking action. Since it is an 8 bit register, the greatest number possible in the accumulator is 255 ($FF).

THE X AND Y REGISTERS

These two registers are also 8 bit ones. They can do most of the things the accumulator can do, but they're usually used as counters or "indexers." For example, the X or Y registers are often incremented something like FOR/NEXT loops. The contents of the register is used as an index or offset to an address. If the register value is "1" then "1" is added to a specified address. Thus, if you wanted to do something with a range of memory without having to keep entering the next memory location, incrementing the X or Y registers is a handy way of doing this. They can also be used for temporary storage, half-way houses and transfer points. In your mnemonic opcodes the letters X or Y are tip-offs that the instruction does something with one of these registers. For example, INX increments the X register and TYA transfers the contents of the Y register to the accumulator. Both registers are 8 bit ones and can only hold a maximum value of 255 ($FF).

THE PROCESSOR STATUS REGISTER (STATUS REGISTER)

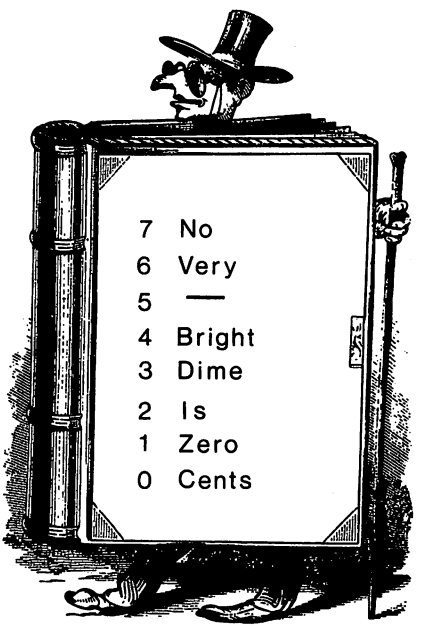
This register is really different from the A, X and Y registers. In fact, it is actually 7 little 1-bit registers packed into one 8 bit byte. (One bit is not used.) Called the 'P' or Status register, it is affected by what various opcodes do within a program. When we discuss branching and looping, we will see more clearly how it works. For now, let's take a look at its make-up.

```
 7  6  5  4  3  2  1  0  ◄-Bit
-------------------------------------------------------------
 N  V  *  B  D  I  Z  C  ◄-Register
```

Before we see what the initials really stand for, here's a way to remember them in order:

No Very Bright Dime Is Zero Cents

Just remember "bright dimes" are worth ten cents, not zero cents. And what's half a dime? It's 5 cents. That'll help you remember bit #5 is not used.



Before we look at the 1 bit registers, you should realize they all have only one of two conditions. You'll remember that a single bit can either be "Ø" or "1." If the register is "1", we say that it's Flag is Up or set. If the content of the bit is "Ø" then the Flag is Down or cleared. That's easy to remember since each individual 1-bit register can only be one or the other - set or cleared. (Remember when you clear out your drawer, it has "Ø" contents.)

Negative Flag

When the results of an operation is "negative" the flag is set. (That means there's a "1" in bit 7.) On positive results, the flag is cleared, meaning a "Ø" is in bit 7. Loading a register (A,X or Y) with a value of less than 128 will clear the N flag; loading a value of greater than 127 will set it. Thus, "negative" refers to values greater than 127 and "positive" to register values less than 128. (It has to do with "two's complement" math, which we will not cover in this book. In other words, don't worry about it.)

oVerflow Flag

This is set in signed arithmetic operations and cleared with a special opcode. Don't worry about it for the time being, for at this stage it won't concern us.

Break Flag

If the last instruction was BRK this flag is set. This is another flag we will not be using much at first.

Decimal Flag

This flag is set and cleared for certain decimal operations with SED and CLD respectively. We will not be concerned with this flag as it is primarily used for determining the type of arithmetic the microprocessor is to use.

Interrupt Flag

This flag is set if there is a hardware interrupt. We won't be going into using this flag since it deals with hardware conditions in your machine.

Zero Flag

If the last result was a "Ø" then this flag is set. This flag will be used a lot by our branching instructions. The crazy thing about this flag is that if the last result in the accumulator was "Ø" then the flag is set which means there's a "1" in bit-1. So that this won't confuse you, just remember that when the last result was zero, the Z register gets all excited and waves a flag.

Setting a flag

Carry Flag

This flag is set when a carry/no-borrow condition arises. Other logical and math operations also will set this flag. You can use it for other things as well since there are opcodes for setting (SEC) and clearing (CLC) the C flag. By using other instructions that read the Carry flag, you can use it in various ways.

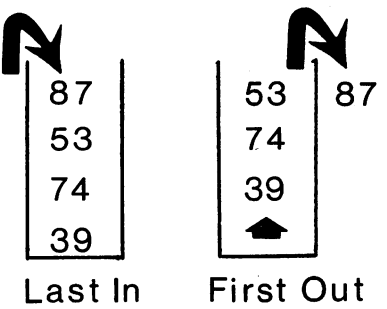
At the beginning level of assembly language programming, many of the Status register conditions will not directly affect us. The biggest problem encountered by beginners with the Status register is when some condition exists they accidentally created. Unexpected results may arise they do not understand. To avoid these unknown sources of problems, we will be very careful in not tackling some more advanced techniques affecting the Status register. However, as you start experimenting on your own (as you certainly *should* do!), you're going to have to go beyond the scope of this book for understanding all the possible circumstances leading to unwanted results. However, you should be aware that the Status register is one place to examine for the cause of unexpected results.

STACK POINTER

This register handles some important information that won't concern you directly at first but is critical to your program. The stack pointer holds the return address for subroutine jumps. In BASIC when your program executes a GOSUB, it needs something that will tell it where it jumped from. This is called the return address. With assembly language, you will be using JSR (Jump to SubRoutine) right away. The stack pointer tells the JSR where to return after it has made the jump. Thus, while all the programmer has to enter is JSR and everything will be handled automatically by the stack pointer, the stack pointer itself is crucial to program's operation.

The stack itself is located in "Page 1" of your memory; locations $Ø1ØØ - $Ø1FF. (Page 2 is from $Ø2ØØ-$Ø2FF, Page 3 from $Ø3ØØ-$Ø3FF, etc.) The stack works like the spring loaded tray dispenser they have in cafeterias. As you remove each tray, a spring

underneath the trays pushes the next one up. If you put a tray on top of the stack, it will be the first one you take off. That means the Last In is the First Off. This is called a LIFO arrangement.



Your stack starts putting values at address $Ø1FF, the top of the stack. If you add another value, the stack pointer moves down to point to the next lowest address, $Ø1FE. If you take one element off the stack, the stack pointer is moved up to point to $Ø1FF. Since the stack stores two-byte addresses, such as $CØØ4, usually we're dealing with two locations on the stack at a time. Thus, $Ø1FF might be $CØ and $Ø1FE, $Ø4. By moving the stack pointer down to point to the most recently added value, the last value entered becomes the first released. (The "last hired" is "first fired.")

As we said, you probably won't be doing much with the stack right away since the most important work of the stack is keeping return addresses. This is done automatically with JSR and other instructions that require return addresses. However, assembly language programmers often use the stack for temporary storage. The stack pointer, which tells the next available location on the stack, starts at $FF and works its way toward $ØØ. In our above example, where the return address of $CØØ4 is using stack locations $Ø1FE and $Ø1FF, the stack pointer points to $Ø1FD as the next location on the stack.

```
Stack
----------
$Ø1FF   $CØ
$Ø1FE   $Ø4
$Ø1FD   ----- ◄- Stack Pointer points here
```

```
Stack Pointer
-----------------
$FF
$FE
$FD ◄- Low byte
.....
$Ø1 ◄- High byte is always $Ø1
```

Because the stack pointer is an 8 bit register, the most it can hold is 255 or $FF. Since it needs a way to deal with values up to $Ø1FF, it needs a high byte of $Ø1. To handle the high byte, $Ø1 is always the high byte of the stack pointer. Thus, even with an empty stack with the pointer pointing to $Ø1FF, the $FF is always combined with the high byte of $Ø1 to arrive at $Ø1FF. In this way, the pointer can handle the entire stack with an 8 bit register.

PROGRAM COUNTER

The program register is a 16-bit register storing the next address to be executed in a program. As we will see in more detail in the next chapter in discussing your Commodore 64's memory, the program counter stores addresses in a Low-Byte, High-Byte configuration. The 16 bit register is actually two 8-bit registers arranged as follows:

Program Counter Low	Program Counter High
7 6 5 4 3 2 1 Ø	7 6 5 4 3 2 1 Ø
LO-BYTE	HIGH-BYTE

As each instruction is executed, the program counter is incremented to the next address where an instruction will be stored. In this way, your computer can execute instructions in the correct order. However, since this is done automatically, you don't have to worry about it.

INPUT/OUTPUT PORT

This port is located at locations $ØØ and $Ø1 in your computer and is the principle difference between the 651Ø and the 65Ø2 microprocessors. It is used for directing input and output (I/O),

and other than suggesting that you stay out of those locations, we're not going to be dealing with this port at all. The $Ø location is the data direction register, and $1 is the I/O port itself.

SUMMARY

It's easy to get tangled up in the registers if you try to do too much too soon. For the most part, your major operations will be with the accumulator, and the X and Y registers. Your program will be affected by the other registers, but as in BASIC where most of the computer's operation are taken care of for you, in assembly language programming, most of the registers take care of themselves. Thus, while its useful to know about the P register's flags, and we will be using instructions that rely on those flags, you don't have to actually "hand-set" the flags. Rather, you simply will be writing opcodes that take care of the flags for you. As you become more advanced, you will be making more direct use of the registers in your programs. To begin, though, you don't have to keep track of everything the registers are doing.

So if you feel a bit lost right now, and you do not think you understand everything about the various registers that work with your microprocessor, don't worry. While assembly language programming requires that you give your computer more instructions, the bulk of what goes on is fairly automatic. Just remember to take things a step at a time when we get into writing actual assembly language programs. With practice and experimentation, what we have covered will become clearer.

CHAPTER 7

MEMORY AND STORAGE

LINE NUMBERS AND ADDRESSES : A COMPARISON

When you write a BASIC program, the line numbers you use are a point of reference. It doesn't matter whether you number your program 1,2,3,4,5,6, etc. or 10,20,30,40,50, etc. or even 1,10,32,113,2000. All you have to do is to make sure that the program statements are in order. In fact, the line numbers are simply a convenient way of ordering your statements so your program knows what comes next. Thus, the following BASIC programs all do the same thing despite the different line increments used:

```
10 PRINT CHR$(147)
20 FOR X = 1 TO 10
30 PRINT X
40 NEXT X
50 END
```

```
5 PRINT CHR$(147)
11 FOR X = 1 TO 10
12 PRINT X
130 NEXT X
2000 END
```

```
1 PRINT CHR$(147) : FOR X = 1 TO 10
3 PRINT X : NEXT X
5 END
```

Not only can we use any ordered (from lower to higher) line numbering system, we can put several statements in the same line. Your BASIC interpreter handles all the details for you. It assigns the BASIC tokens, knows what is a token and what is a value, and sticks the program in an area reserved for BASIC programs.

With assembly language programming, the first decision you make is *where* to put your program. If you put it in the wrong place, it'll bomb. For example, if you want to execute your machine program from a BASIC program, you'd better not put your machine code in the area used by BASIC. Likewise there are other areas in your computer that will conflict with your program, or simply will not accept the code you enter.

Once you decide where to put your program, you must use contiguous addresses. That means, if your last instruction used addresses 49152, 49153 and 49154 ($C000-$C002), the next instruction *must* begin at 49155 ($C003). You are not using numbers simply as a way of ordering your commands as in BASIC, you are actually stuffing information into addresses that will be executed sequentially. FORTUNATELY, YOUR ASSEMBLER WILL KEEP TRACK OF THIS FOR YOU. All you have to do is to make sure that the area you're using is clear. So while it is imperative to have a clear spot in RAM to put your program, the addresses used from that point on will be handled by your assembler.

What confuses most beginning assembly language programmers is the different ways that various assemblers handle this. Let's look at the three covered in this book to see how each deals with this.

Kid's Assembler : No Line Numbers

Since this assembler is meant to teach you about what's happening to your code as you enter it, the actual addresses are given as you enter your opcodes and operands. The addresses are given in decimal values so that you can see how many bytes of memory each

instruction uses. For example, when using the Kid's Assembler you would see the following:

```
ADRS     OPCODE  OPERAND
-------------------------
49152    JSR     $E544
49155    LDA#    Ø
49157    TAX
49158    STX     $DØ21
etc.
```

In the above example the first instruction used 3 bytes at addresses 49152, 49153 and 49154. Therefore, the next available address is 49155, which you can see in the second line. The next instruction used only 2 bytes, 49155 and 49156, and the third instruction, only a single byte. Thus, while the increments from one instruction to the next may be uneven, you can get a clear idea of where your code is being stored and how much memory is being used.

Merlin64

When you use the editor in Merlin, you get line numbers incremented by one as you enter your program. This system makes it easier to keep track of everything, and since you can insert code between the lines, you don't have to worry about having spaced between line numbers. With the powerful editing features in Merlin64, you can manipulate and change your opcodes and operands. However, until you learn how many bytes each instruction uses in the different addressing modes, you won't know how much memory you used until you ASseMble your code. At that time you will see the actual addresses and the number of bytes you used. You don't have to worry about addresses, other than where to start placing your code, since like the Kid's Assembler, this is automatically done for you.

Commodore Development System

When using EDITOR64, you *should* treat your line numbers the same way as you would a BASIC program. This is because you will need room to insert lines between line numbers. If you run out of

room between lines, you can reNUMBER your lines (fortunately) to insert code. These line numbers, however, are *not* the addresses you in which your code is stored. Rather, they simply represent the sequence in which your instructions will be assembled. Once you use your ASSEMBLE64 program, you can see the addresses where the code is stored.

ROM AND RAM MEMORY

If you don't already know it, there's an important difference between ROM and RAM memory. Basically, ROM memory is "iron plated" and what is there, stays there. Your BASIC ROM, for example, is always the same no matter how many times you turn your computer on and off. (You can move ROM routines into RAM and change them in RAM. However, this has no effect on what's in ROM.) RAM, on the other hand, can be changed simply by entering new values in the RAM addresses. You might imagine your computer's memory as a book with only some of the pages with print. Those pages with print have information that stays on those pages. It also has some blank pages where different material can be added or changed. The blank pages are like RAM and the printed pages like ROM. However, even with the blank pages, there are certain ones that are reserved for certain things you will usually want on those pages. For example, while BASIC is in ROM, it loads into RAM in certain locations. You can change what has been put in RAM, but usually you leave it alone. That leaves certain other pages that are almost always blank, and you can write whatever you want in them. These are the pages we will use to store our assembly language programs.

Your Commodore 64 can access more than 64,ØØØ RAM addresses, but we're going to concentrate on basic 64K RAM configuration. Some parts of this RAM are normally used for the information stored in ROM. Thus, while we will refer to them as ROM areas, in fact, they can be used as RAM. (Think of these areas as having ROM routines loaded into them from the ROM chips.) First, lets look at a "map" of how your Commodore 64 defaults memory set-up. This will be used in our programming considerations even though some assemblers rearrange memory to provide more space for machine language programs.

STANDARD MEMORY ALLOCATION

$E000-$FFFF 57344-65535	8K Kernal ROM
$D000-$DFFF 53248-57343	4K I/O or Character ROM
$C000-$CFFF 49152-53247	4K RAM
$A000-$BFFF 40960-49151	BASIC ROM or ROM Plug-in
$8000-$9FFF 32768-40959	8K RAM or ROM Plug-in
$4000-$7FFF 16384-32767	16K RAM
$0000-$3FFF 00000-16383	16K RAM

First 32K

At first glance, you might look at those two 16K blocks of RAM at the bottom and decide that's the place to store your machine code. As it happens, this is not the case. First of all, you won't be doing any programs that use 32,000 addresses for a while, and if you did, you'd run into the area where BASIC is stored as well as all kind of other things you don't want to crash into.

8K RAM or ROM Plug-in

Right above the second block of RAM is an 8K block that is used either as RAM or a ROM plug-in. This is a good place to store your

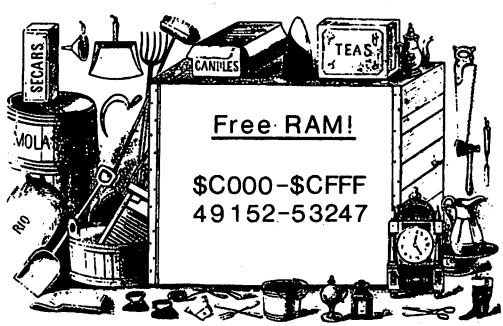
programs if you don't use a ROM plug-in. However, you may want to write programs that others who do use plug-in ROMs will use. If you store your programs here, they may be over-ridden or conflict with a plug-in. For example, I have a plug-in utility program called "Vic-Tree." I use it all the time for easily accessing my disk and editing BASIC programs. If I have a machine code program in the area used by the "Vic-Tree", as soon as I initiate the ROM with SYS 32768, I blow out the machine code stored there.

8K BASIC ROM Area

The next area of RAM, $A000-$BFFF, is where your BASIC ROM loads in. Programs stored here will conflict with your BASIC operations. Since you will often want to SYS a machine routine from a BASIC program, any code stored here will be wipe out your BASIC.

4K RAM

Next, we come to a nice clean-looking 4K area of RAM beginning at $C000 (49152). This is the area I like to use the most since there's plenty of room to write programs, and it doesn't conflict with anything else. It's above the BASIC ROM area, out of the way of plug-ins, and very far away from where your BASIC instruction storage begins ($800/2048).



4K I/O or Character ROM

Leave this area alone for now since your characters are stored here or used for your I/O.

8K Kernal ROM

This area is a good place to visit with JSR since it has so many useful subroutines you will want to use. However, if you store your program here, you'll conflict with the Kernal. Think of this area as a tool box. You build your program somewhere else but use all the tools in this area.

Some nooks and crannies

Now you may be thinking all you have is a crummy 4K for your programs. That's almost true if you want to use your BASIC and have room for plug-ins. However, there are some other places you can use while keeping all your other goodies in tact. Probably the favorite for assembly language programmers is the cassette buffer from $33C-$3FB (828-1019). That gives you only 192 bytes, but we're going to be using short routines to get started; so there's more than enough room here. The only problem is that if you have a cassette recorder, you'll crash into some routines used by the cassette. Likewise, there are little places you can find that are unused or so little used that you're probably safe. For example, there are eight bytes available from $334-$33B, Sprites 13-15 from $3CF-$3FF as well as other little hidey holes. However, its not worth the bother to even worry about these locations until your programs get really big. By that time, though, you'll probably want a reconfiguration of memory to use everything taken up by BASIC.

For the time-being there's plenty of room to work in the 4K area of RAM. In fact, we'll be able to stick dozens of little routines there simultaneously that we can SYS from BASIC. After that, the 8K block of RAM beginning at $8000 is available unless you use a plug-in ROM a lot. (Merlin64 defaults to this area since the assembler/editor uses $C000. However, there's no problem in having your programs executed from the $C000-$CFFF area once Merlin is out of memory. We'll discuss this more a little later.)

MINI-MONITOR

To look at the contents of your ROM and RAM, monitors are handy. In previous chapters, I suggested you write your own

monitor, and I still think you should. However, to get you started on your monitor, the following "mini-monitor" examines the contents of your memory for you. Addresses and their contents are presented in hexadecimal and decimal. In this way, you can see where you have free RAM and where there is information stored. Also, as you start learning the hexadecimal machine opcode values, you can actually read the routines in your memory. In the next section we will be discussing the high-byte / low-byte arrangement of address storage, and this too you will be able to see. Moreover, since the mini-monitor is written in BASIC, it will be able to examine itself. Take a look at the code being stored beginning at address $800 (2048).

MINI-MONITOR

```
10 PRINT CHR$(147)
20 PRINT "BEGINNING ADDRESS OR {RETURN} FOR
NEXT"
30 INPUT"ADDRESS. PRESS 'Q' TO QUIT ";AD$
40 IF AD$ = "Q" THEN END
50 AD = VAL(AD$)
60 FOR K = AD TO AD + 15: N = K
70 HB = INT(N/256)
80 LB = N-INT(N/256)*256
90 FOR X = 1 TO 2
100 IF X = 1 THEN N = HB
110 IF X = 2 THEN N = LB
120 N% = INT(N/16) :GOSUB 250
130 N% = N-N%*16:GOSUB 250
140 IF X = 1 THEN H1$ = HEX$ : HEX$ = ""
150 IF H1$ = "0" THEN H1$ = "00"
160 NEXT
170 HEX$ = H1$ + HEX$
180 HEX$ = "$" + HEX$ :PRINT HEX$;"-";:HEX$ = ""
190 N =  PEEK(K)
200 N% = INT(N/16) :GOSUB 250
210 N% = N-N%*16:GOSUB 250
220 PRINT HEX$;" ";K;"-";PEEK(K) : HEX$ = ""
230 NEXT
```

```
240 AD$ = STR$(K) : GOTO 20
250 REM **********************
260 REM CONVERT DECIMAL TO HEX
270 REM **********************
280 HEX$ = HEX$ + CHR$(48 + N% + 7 * ABS(N% > 9))
290 RETURN
```

You enter your starting address in decimal. Once you've done that, just press RETURN to look at the next block of memory, or enter a new starting address. The hexadecimal and corresponding decimal values are displayed on the same line to help you get acquainted with going from one number system to the next.

Since your mini-monitor simply looks at hex and decimal values, it isn't much of a monitor. See if you can change it to do some or all of the following:

1. Move blocks of code from one area to another.
2. Allow starting addresses to be entered as either decimal or hexadecimal.
3. Change memory values. (A simple POKE subroutine will do that.)
4. Display mnemonic opcodes for machine opcode. (This is tricky since you have to distinguish opcodes from address values, but it can be done. The trick is in knowing the number of bytes each opcode uses.)

This can also be used as a hex-decimal conversion program for converting useful Kernal addresses to decimal so that you can SYS them from BASIC. In fact, one of the first areas you will want to explore is from 57344-65535 ($E000-$FFFF).

BACKWARD NUMBERS : LOW-BYTE / HIGH-BYTE STORAGE

In discussing the numbering systems used in your computer, we arranged the high-byte and low-byte in the order we read a number. For example, a common subroutine in your kernal is at location

$E544. You've seen this in our example programs, and we'll be using it a lot more. Broken down into a high-byte and low-byte, it would look like the following:

HIGH BYTE	LOW BYTE
$E5	$44

However, your Commodore 64, along with other 65Ø2 based microcomputers, store two-byte numbers in a low-byte / high-byte configuration. Therefore, in your computer's memory, it would look like this:

LOW BYTE	HIGH BYTE
$44	$E5

Everything else is in the expected order, but addresses are stored backwards. For instance, the following instruction,

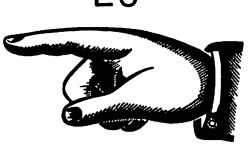
```
JSR          $E544
```

would be stored in three addresses as

```
$CØØØ  2Ø
$CØØ1  44 ◄-Low byte
$CØØ2  E5 ◄-High byte
```

44

E5



The '2Ø' is the hexadecimal machine opcode for JSR. It's just where we would expect it to be since it is the first in our assembly instruction and is at the first address. However, the second or low byte, 44, of $E544 comes first, followed by the E5. If we had made our JSR to $22, our code would be arranged as follows:

```
$C000   20
$C001   22◄- Low byte
$C002   00 ◄- High byte
```

And a JSR to $FF00 would look likes this:

```
$C000   20
$C001   00 ◄- Low byte
$C002   FF ◄- High byte
```

So if you see a number like the following:

```
1A   09
```

Just switch it so it is:

```
09   1A
```

And you get $91A.

BYTES, OPCODES AND ADDRESSING MODES

We've discussed the fact that different opcodes use different amounts of memory, and the same mnemonic opcodes in different addressing modes use different numbers of bytes. As we get into the actual use of opcodes in the next chapter, you will be better able to see how this works. For now though, we should preview this a bit.

The first thing to remember is that an opcode uses only 1, 2 or 3 bytes of program memory. Let's examine each in terms of bytes used.

1 Byte

The most economical opcodes are those using only a single byte. For example, we've seen RTS. It has no operand, and so it is said to be an implied opcode. (Implied addressing.)

2 Bytes

Opcodes that can only handle 1 byte operands take up two bytes. One byte is for the opcode and the other for the operand. Any instruction that can have a maximum operand value 255 ($FF) is a two byte instruction. For example, in the immediate mode, LDA can only have a maximum operand value of 255 ($FF).

3 Bytes

Opcodes whose operand is a non-zero page address have three bytes. One byte is used for the opcode, one for the low-byte of the address and one for the high-byte. For example, LDA in the absolute addressing mode uses three bytes.

SUMMARY

As you practice and experiment, the material we have covered will become easier. In the next several chapters we will be putting to work the knowledge we have developed. Thus, what has been abstract will become concrete and give you a new level of understanding. You will make mistakes, but whereas an uninformed programmer does not understand why he made a mistake, you will. For example, you may accidentally load your program into $8ØØ instead of $8ØØØ. Since $8ØØ is the beginning of BASIC, any BASIC program you try to execute after loading a machine language routine in that location will bomb. Because you now know something about how memory works, you will be able to spot the problem and correct it.

CHAPTER 8

JUMPING IN

WHERE TO STICK YOUR PROGRAMS

Just in case you skipped our discussion of program placement, we'll quickly review both the procedure for assigning the starting address and the recommended places to start. Since each assembler is a little different in this regard, read the section that applies to the assembler you're using.

Auto-Placement With Kids' Assembler

If you're using the Kids' Assembler, the default starting address is 49152 ($C000.) When you enter the editor/assembler all you have to do is press RETURN, and your program is stored beginning at 49152. If you want another address, just enter the *decimal* value for the address and press RETURN.

ORG Pseudo-Opcode In Merlin64

If you do not specify a beginning address with ORG in your first line; then it will default to $8000 (32768). This is fine if you do not use plug in ROMs, and you can test your programs while Merlin is still in memory if you use this address. However, if you want your

programs in $CØØØ or some other area outside of the plug-in ROM location, then use the following format:

```
LABEL     OPCODE    OPERAND  ;COMMENT
-------------------------------------------------------------------------
          ORG       $CØØØ
```

This should be your first line of code in the programs we will be covering in this book. Once you're more advanced you can do more with ORG in different places in your program.

Commodore Assembler's * = function

If you use the Commodore EDITOR64, your first line should be * = to some clear area of memory. Since the default address is zero ($ØØØØ), you must begin your programs with the * =.

```
LABEL     OPCODE    OPERAND  ;COMMENT
-------------------------------------------------------------------------
          * = $CØØØ
```

Since the Commodore 64 Macro Assembler Development System uses special loader programs to put your assembled code into memory, be careful in choosing your loader. Since our examples will stick with the 4K of RAM beginning at $CØØØ (49152), you should use the LOLOADER. It loads at $Ø8ØØ. However, since the HILOADER loads at $C8ØØ, which will be above our example programs, you can use it as well.

One final thing about the Commodore editor/assembler; you must end your programs with .END in the OPCODE field. Neither the Kids' Assembler or Merlin's Apprentice understands this pseudo-opcode; so only use it with the Commodore system.

VISITING BUILT-IN SUBROUTINES WITH JSR.

At the top of your memory is the Commodore-64 Kernal. A visit to the Kernal with the JSR instruction will execute the subroutine at the addresses specified in the operand and return to your program. For example, one subroutine we'll want to use is called CHROUT.

It outputs a character to a specified channel, usually the screen. The CHROUT routine is located at $FFD2 (65490). Likewise, at $E716 (58159), there is a subroutine to output the contents of the A register (accumulator) to the screen. Both subroutines can do the same thing, but each requires different preparation routines we will learn. In Appendix D there is an extended listing of Kernal subroutines. As we go along, we'll introduce some other handy subroutines that are built into your Commodore-64.

The best way to envision the JSR instruction is as a GOSUB. However, unlike BASIC where you must write your own routines, the JSR goes to a built-in subroutine. (Of course you can JSR to subroutines you've written yourself, but that will come when you're more advanced.) For the time being, think of the JSR in terms of built-in subroutines that do all kinds of things for you. In fact, the JSR instruction is actually simpler than GOSUB since with BASIC GOSUBs you have to prepare your own subroutine. Just imagine a really nice person at Commodore who spent a lot of time writing complex subroutines for you to make assembly language programming easier. The following is a sampling of some handy subroutines to visit with JSR:

HEX	DECIMAL	FUNCTION
$E544	58692	Clear screen
$E566	58726	Home cursor
$E716	59159	Output to screen
$E891	59537	Perform RETURN
$E8E7	59624	Scroll screen
$E9FF	59903	Clear screen line
$FF9F	65439	SCNKEY-scan keyboard
$FFCF	65487	CHRIN-get char from input channel, usually keyboard
$FFD2	65490	CHROUT-output a character
$FFE4	65508	GETIN-get a character, usually from keyboard
$FFF0	65520	PLOT-set cursor location

Most of the built-in subroutines expect some value in the accumulator or some preparatory set of instructions. For example, a

JSR to $E516 (output to screen) sends whatever is in the A register (accumulator) to the screen. Other subroutines, such as CHROUT expects a channel defined by the CHKOUT subroutine, and in turn, CHKOUT expects the OPEN subroutine to prepare the channel. We'll get to these as we accumulate more instructions.

To get you going, we'll do some simple JSR's to subroutines expecting nothing. However, before we do that, we'd better look at RTS.

GETTING OUT WITH RTS

Probably the most frequent cause of Computer "lock-up" is the failure to end a routine with RTS. If you execute a machine language program from BASIC in either the immediate mode or from a program with SYS, control is taken over by your machine language program. The program executes instructions one address at a time. If there is not an instruction to tell the program to ReTurn from Subroutine (RTS) then it merrily goes on to the next address and instruction. This can be a real problem if the area where you're programming has "garbage" in it, either in the form of the remnants of another machine language program or other code that somehow got placed there. For example, the following might be a simple routine to clear the screen that forgot to include an RTS:

```
$C000 JSR  $E544
Garbage from here on
$C003 LDA  #0 <- Remnants from other program
$C005 STA  $D021
$C008 LDA  #144
$C00A JSR  $E716
More junk...
```

Your JSR to $E544 sure cleared the screen for you, but the "garbage" turned your background and character color to black. So instead of having a nice clear screen and control over your computer, you have black on black with no visible cursor!

Now that we have JSR and RTS, let's write some simple programs:

```
LABEL/ADRS OPCODE OPERAND ;COMMENT
----------------------------------------------------------------------------
           ORG     $C000   ;Merlin only
         * = $C000           ;Commodore only
49152      JSR     $E544
49155      RTS
           .END              ; Commodore only.
```

After entering RTS on the Kids' Assembler, enter 'Q' for 'quit.' As our listings become more complex, we will have separate listings for the Kids' Assembler and a general listing for other assemblers. You will have to remember to include the ORG or * = pseudo-opcodes (or whatever your assembler uses to indicate the start address of your program) and to end it with .END on your Commodore assembler.

Now, you've seen that little program elsewhere in this book, and by now you should not only know how to clear your screen, but how to get back control of your computer once you've done that. Just to see if you understand the concept of JSR, see if you can write a program that will Home the cursor but not clear the screen. (HINT: Find the address of the Home cursor routine.) If you can do that, go on to the next section.

LOADING UP WITH LDA

Your accumulator is the work horse of assembly language programming. Its contents, which can be from $Ø-$FF (Ø-255), are used by other instructions. Furthermore, many of the built-in subroutines expect something in the accumulator. One way to get something into the accumulator is with the LDA (LoaD Accumulator). For example, try the following little program:

GENERAL

```
LABEL          OPCODE  OPERAND  COMMENT
-------------------------------------------------------------------
               ORG     $C000    ; Merlin
                                  Assembler
               * = $C000        ; Commodore
                                  Assembler
-------------------------------------------------------------------
               JSR     $E544    ; Clear screen
               LDA     #88      ; Load the ac-
                                  cumulator with
                                  the decimal
                                  value 88
               JSR     $E716    ; Output to
                                  screen
               RTS
               .END             ; Commodore
                                  assembler
```

KIDS' ASSEMBLER

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------
49152     JSR       $E544
49155     LDA#      88
49157     JSR       $E716
49160     RTS
```

Note the the presence of the pound (#) sign right before the 88 (or at the end of the LDA in the Kids' Assembler.) That means the LDA is in the Immediate mode. The value in the operand is actually loaded into the accumulator.

We'll discuss addressing modes more in a bit, but let's see what the program does. When you assemble and run the program, you will see an 'X' printed in the upper left hand corner of your screen. The "output to screen" subroutine at $E716 took the value in the accumulator, and placed it on the screen. Since the 88 is the ASCII value of the letter X, that's what was output to the screen. Appen-

dix J has a complete listing of the various values for your Commodore Character Set.

The following example loads different values into the accumulator and prints my first name, "BILL". (Remember the ORG on Merlin and *= and .END on your Commodore Assemblers respectively.)

GENERAL

```
LABEL     OPCODE    OPERAND   COMMENT
-------------------------------------------------------------------------
          JSR       $E544
          LDA       #66       ; B
          JSR       $E716
          LDA       #73       ; I
          JSR       $E716
          LDA       #76       ; L
          JSR       $E716
          JSR       $E716
          RTS
```

KIDS' ASSEMBLER

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------------
49152     JSR       $E544
49155     LDA#      66
49157     JSR       $E716
49160     LDA#      73
49162     JSR       $E716
49165     LDA#      76
49167     JSR       $E716
49170     JSR       $E716
49173     RTS
```

Now wait a minute! There are only three LDA's and yet the word BILL has four letters. What's going on? There is nothing in the program that affects the contents of the accumulator. Some subroutines or other actions can scramble the accumulator, but generally what was last loaded with LDA stays there. Therefore, since BILL has two L's together, the value for the letter L (76) was still there after the first one had been output to the screen. To see if you understand everything so far, see if you can change the program to write your own name to the screen.

Loading the accumulator LDA

IMPLIED, IMMEDIATE AND ABSOLUTE ADDRESSING MODES

The program that printed a letter to your screen used three addressing modes, implied, immediate and absolute. The implied mode occurs when no operand exists. RTS is in the implied addressing mode. Instructions in the implied mode only take one byte of memory since all it needs is a single machine opcode.

The immediate addressing mode is signaled by the pound sign (#), usually the first character in the operand. (As explained in Chapter 2, the Kids' Assembler attaches the addressing mode to the end of the opcode.) This means that the contents of the operand are loaded, stored, compared or in some other way acted upon by the operand. Thus, LDA #88 (or LDA# 88), took the value of the operand and put it in the accumulator. Operations in the immediate mode take two bytes; one for the opcode and one for the operand, which can be no larger than $FF (255).

Finally, we used the absolute addressing mode with our JSR instruction. The absolute addressing mode refers to the *address* in the operand, not its value. Since JSR only works in the absolute mode, we don't have to concern ourselves with it. However, LDA also has an absolute mode. If we had written,

instead of loading the accumulator with the value 88, it would have loaded the accumulator with the value stored in address 88 ($58). If you want to see the difference in results, just remove the pound (#) sign from the operand (or the opcode in the Kids' Assembler.) Since locations 87-96 ($57-$6Ø) are a miscellaneous numerical storage area, there's no telling what you'll get. (Go ahead and try it just for fun.)

To summarize the addressing modes we've used so far, keep the following in mind:

IMPLIED MODE. Uses single byte and no operand.

IMMEDIATE MODE. Uses actual value in operand, either decimal or hexadecimal, and uses two bytes.

ABSOLUTE MODE. Uses value in the *address* in operand. Three bytes of memory are used; one for the opcode, and one apiece for the low and high bytes of the address.

= = A COMMON MISTAKE = =

Sooner or later you will make the mistake of mixing up the immediate and absolute modes. You will put in LDA 2ØØ when you meant to write LDA #2ØØ (LDA# 2ØØ on the Kids' Assembler.) Don't worry, it'll happen, believe me. Now that you know it will happen, you also know a common bug in assembly language programs. All you have to do is to add the pound (#) sign and your results will be what you expected in the first place. (Also, you'll wish you had an assembler package with a good editor!)

At this point, we're starting to get somewhere. We've used assembly language commands interactively. That is, what we did with one opcode instruction, affected another. That wasn't too difficult, was it?

STORING WITH STA

In BASIC programming you define variables with statements such as;

```
A = 45
```

In essence, you are storing the value 45 in a variable called A. In your machine language interpreted BASIC program, the variable A is an address in memory; so that whenever A is accessed, the contents of the address for A is accessed.

In assembly language programming, you actually put a value into a certain address. Sometimes you will use an address that you know is empty and is simply a handy place to store a value to be used later in your program. Very often, though, you will store a value in an address that activates a process. This will either supply a value for a built-in subroutine or an address that stores a color or character on your screen. Let's look at each of these uses with the STA instruction.

In the absolute mode STA stores the accumulator's value in a specified address. For example, if you LDA with a value of 1Ø, and then STA $33C, the value 1Ø will then be stored in location $33C.

Storage in Empty, Unused RAM

In planning an assembly language program, you not only have to plan for space for your program, you also have to plan for space for your variables. For example, if your program uses 3Ø bytes for the program itself, it may use another 3Ø bytes, or even more, for variable storage. For example, you may use the addresses from 49152-49181 ($CØØØ-CØ1D) for your machine instructions and 492ØØ-4923Ø to store your variables. Therefore, you need to plan for 6Ø bytes of memory. The following program loads (LDA) a value into the accumulator in the immediate addressing mode (#), stores the value at an unused address (STA), clears the accumulator by loading it with zero, (LDA #Ø), and then loads the accumulator from the absolute mode (LDA) from the address it first stored the

value. To show you what it did, it prints the character for the ASCII code in the accumulator with JSR $E716.

GENERAL

```
LABEL           OPCODE     OPERAND    COMMENT
-------------------------------------------------------------------------
{MERLIN         ORG        $C000}
{COMMODORE      * = $C000}
                JSR        $E544      ; CLS
                LDA        #$43       ; ASCII 'C'
                STA        $C050
                LDA        #$0        ; Immediate
                Mode
                LDA        $C050      ; Absolute
                Mode
                JSR        $E716
                RTS
{COMMODORE      .END}
```

KIDS' ASSEMBLER

```
ADRS      OPCODE   OPERAND
-------------------------------------------------------------------------
49152     JSR      $E544
49155     LDA#     $43
49157     STA      $C050
49160     LDA#     $0
49162     LDA      $C050
49165     JSR      $E716
49168     RTS
```

In the above example, we used only hexadecimal values in the operand. This was to show you that both the pound sign (#) and dollar sign ($) had to be included in the operand in standard assembler format. Before you run the program, see if you can guess the character to be printed to the screen.

STORAGE IN 'SOFT-SWITCH' ADDRESSES

A 'soft-switch' is a switch that is activated by the software. In your Commodore 64, there are many such locations. For example,

the background color of your screen is determined by the value in 53281 ($D021). At the end of Chapter 1, we showed all of the values to be POKEd into 53281 to give you the desired background color on your TV or monitor. When you use a soft-switch address to STA a value, it is the same as POKEing that location in BASIC. For example, to turn your background to black, you would STA the accumulator value in $D021. The following little program shows you how:

GENERAL

```
LABEL           OPCODE   OPERAND COMMENT
----------------------------------------------------------------------
{MERLIN         ORG      $C000}
{COMMODORE      * = $C000}
                LDA      #$0
                STA      $D021
                RTS
{COMMODORE      .END}
```

KIDS' ASSEMBLER

```
ADRS        OPCODE    OPERAND
-----------------------------------------------------------------------
49152       LDA#      $0
49154       STA       $D021
49157       RTS
```

There are a lot of pointers, flags and registers that can be treated as soft-switches, and simply with LDA and STA instructions, you can change them to what you want. The following is a list of some soft switches and pointers we will be using: (Later on we will deal with the many registers that affect your sprites and sound.)

SOFT SWITCHES AND POINTERS

HEX	DECIMAL	FUNCTION
$2B-$2C	43-44	Start of BASIC
$2D-$2E	45-46	Start of BASIC variables
$286	646	Current character color

$28A	650	Repeat all keys if $80
$D020	53280	Border color 0-15 ($0-$F)
$D021	53281	Background color 0, 0-15 ($0-$F)
$D022	53282	Background color 1, 0-15 ($0-$F)
$D023	53283	Background color 2, 0-15 ($0-$F)
$D024	53284	Background color 3, 0-15 ($0-$F)

Let's make a simple and practical program. In fact, let's make two. One will make all your keys repeat, and the other will turn off your key repeat. We'll store one program at 49152 and the other at 49200. When you SYS 49152, all your keys will repeat, and when you SYS 49200, the repeat will be turned off. These programs can be loaded simultaneously while you program in BASIC. You might want to turn on the repeat function if you're working with keyboard graphics and turn it off if you're working with text. Save the first program under the name ON and the second OFF.

GENERAL - "ON"

```
LABEL           OPCODE    OPERAND   COMMENT
-------------------------------------------------------------------------------
{MERLIN}        ORG       $C000
{COMMODORE}* = $C000
                LDA       #$80
                STA       $28A
                RTS
{COMMODORE} .END
```

KIDS' ASSEMBLER - "ON"

```
  ADRS    OPCODE OPERAND
-------------------------------------------------------------------------------
  49152    LDA#    $80
  49154    STA     $28A
  49157    RTS
```

GENERAL - "OFF"

```
LABEL          OPCODE   OPERAND  COMMENT
-------------------------------------------------------------------------
{MERLIN}       ORG      $C030
{COMMODORE}* = $C030
               LDA      #$0
               STA      $28A
               RTS
{COMMODORE}.END
```

KIDS' ASSEMBLER - "OFF"

```
 ADRS    OPCODE OPERAND
-------------------------------------------------------------------------
 49200    LDA#    $0
 49202    STA     $28A
 49205    RTS
```

When you want to use the programs, first load them into memory either with LOAD "PRG NAME",8,1 (for Merlin and Kids' Assembler files) or with the LOLOADER program for files created with the Commodore ASSEMBLER64. When you want your keys to repeat, just enter SYS 49152, and when you want to turn off the repeat function, enter SYS 49200. (Congratulations, you've just written your first utility programs in assembly language!)

= = AUTO-LOADING MULTIPLE PROGRAMS = =

When you load more than a single machine language PRG file from the immediate mode in BASIC, you have to enter NEW after each load. However, Guy Grotke, in his book *Intermediate Commodore 64,* shows a way to load as many programs as you want with a BASIC loader program. The essential problem with multi-loading from a BASIC program is that each time you load a file, the pointer goes back to the beginning of the BASIC pro-

gram. As a result, you get caught in an endless loop since the program will simply keep re-loading the first program. However, since the BASIC program preserves its variables, you can arrange things so that after loading the first program, it will go on to load the next one. For example, the following programs show how to load the ON and OFF files saved with Merlin's Apprentice and the Kids' Assembler:

ON/OFF LOADER - MERLIN

```
10 IF X = 0 THEN X = 1 : LOAD "ON.O",8,1
20 IF X = 1 THEN X = 2 : LOAD "OFF.O",8,1
```

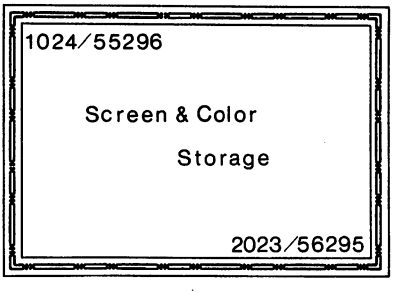
ON/OFF LOADER - KIDS'S ASSEMBLER

```
10 IF X = 0 THEN X = 1 : LOAD "ON 49152",8,1
20 IF X = 1 THEN X = 2 : LOAD "OFF 49200",8,1
```

What happens is that the first time through the program, the IF/THEN statement in line 10 is evaluated as 'true' and so the variable X is incremented by 1 and the first file is LOADed. After the LOAD, the program does *not* go to line 20, but because the pointers were reset by the first LOAD, it starts all over again. However, because X is now 1 instead of 0, Line 10 is ignored and the program proceeds to line 20, evaluates the IF/THEN statement as true, increments X by 1 and then LOADs the second program. By adding more lines and incrementing the X variable, you can LOAD as many PRG files as you want. (If you have the Commodore assembler package, you will have to write and save your programs as PRG files from one of the MONITOR programs. Otherwise, using ASSEMBLER64, your programs are saved as SEQ files and they will have to be loaded with LOLOADER or HILOADER 'by hand.') With that handy little program, whenever you want your repeat key utility programs in memory, just RUN the ON/OFF LOADER.

STORAGE ON YOUR SCREEN

A final place to STA values is right on your screen! Your screen memory 40 x 25 matrix is located from 1024-2023 ($400-$7E7). Overlaying the screen memory on the same matrix is the Color Memory from 55296-56295 ($D800-$DBE7). Appendices H and I provide maps of screen and color addresses to make it easy for you to see where characters will be stored on the screen. When you store a character on the screen, you also have to store a color or you won't be able to see the character. (On the first Commodore 64's it was unnecessary to store the color to see the characters.) The color codes are the same as the ones for background and border colors, 0-15 ($0-$F), and the character codes are the ASCII values found in Appendix J.



To align the character and color, you use the offset 54272. By adding the offset to your screen memory address, you will have the correct address of your color under the character. For example, if you look at the Screen Memory Map in Appendix H, location 1484 is just about in the middle of the screen. Add 54272 + 1484 to get the correct color address of 55756. If you look at the Color Memory Map in Appendix I, you will see that value also to be about in the middle of your screen. The following shows you the correspondence between the screen and color memories:

Screen	Color
1024	55296
1025	55297
1026	55298

1027	55299
1028	55300
.....	.....
2023	56295

Now, using LDA and STA, let's write a program that will put something on the screen.

GENERAL

```
LABEL           OPCODE  OPERAND  COMMENT
-------------------------------------------
{MERLIN}        ORG     $C000
{COMMODORE}     * = $C000
                JSR     $E544
                LDA     #0       ; Black
                STA     55296    ; 1st Color
                                    Address
                LDA     #88
                STA     1024     ; 1st Char
                                    Address
                RTS
{COMMODORE}     .END
```

KIDS' ASSEMBLER

```
ADRS    OPCODE  OPERAND
-----------------------------------------------------------------------------
49152   JSR     $E544
49155   LDA#    0
49157   STA     55296
49160   LDA#    88
49162   STA     1024
49165   RTS
```

We've used the decimal value 88 before, and when we printed it to the screen, we got an 'X.' This time, though, we get a 'spade' character. If you look in Appendix J, the Set 1 character for 88 is

the spade. Set 2 is the X. Thus, you learned that when you LDA a value and JSR $E716, the character printed is Set 2, but if you STA the value of the accumulator at a screen address, you get the character from Set 1. The color stored at the corresponding color address give the character its color, not the background.

SUMMARY

With just a few opcodes, we've already been able to output characters to the screen and even write a little utility program in assembly language. The JSR, RTS, LDA, and STA opcodes are heavily used in all assembly language programing, and so you're off to a roaring start. We also learned about three different addressing modes, and so in addition to the four opcodes, actually have an additional opcode since we learned to use LDA in the immediate and absolute modes.

In the next chapter, we going to learn some new opcodes that will add power to your programming. Right now, what we're doing is equivalent to using a huge number of PRINT and POKE statements in BASIC. However, since we can access built-in subroutines with JSR, we're actually further along in assembly programming than we would be in learning BASIC. By taking each opcode a step at a time, we are able to do a lot with a little; so be patient and forge ahead!

CHAPTER 9

USING THE X AND Y REGISTERS

HOW TO USE TO X AND Y REGISTERS

In addition to your A register, you have two more heavily used registers called X and Y. Like the opcodes referring to the A registers that have an 'A' in their name, such as LDA and STA, opcodes with X and Y refer to the X and Y registers. For example, to load the X register, the opcode LDX is used. (Come on now, what would the LoaD the Y register be?)

There are many uses of the X and Y registers, as there are for the accumulator. Some subroutines read the X and Y registers to calculate certain results. We saw in the last chapter how the "output to screen" subroutine at $E716 took the ASCII value stored in the accumlator and printed a character to the screen. Likewise the PLOT subroutine at $FFFØ reads the X and Y registers to plot the row and column position of the cursor. For example, if we wanted to move the cursor to the middle of the screen on a 4Ø by 25 matrix, we would want to specify a location of about 12/19. Let's see how we can use the X and Y registers to output the contents of the A register to the middle of the screen.

GENERAL

```
LABEL           OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{MERLIN         ORG     $C000}
{COMMODORE      * = $C000}

                JSR     $E544
                LDX     #12     ; Row number
                LDY     #19     ; Column
                                  number
                CLC             ; Clear the C
                                  flag
                JSR     $FFF0   ; PLOT
                                  subroutine
                LDA     #88
                JSR     $E716   ; Output to
                                  screen
                RTS
{COMMODORE      .END}
```

KIDS' ASSEMBLER

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDX#    12
49157   LDY#    19
49159   CLC
49160   JSR     $FFF0
49163   LDA#    88
49165   JSR     $E716
49168   RTS
```

First of all, we did something sneaky! We included an opcode not yet covered, CLC. The CLC instruction CLears the Carry flag. The reason we did that is because the PLOT subroutine at $FFFØ can be used to either read the X/Y position of the cursor or to set it. If the Carry flag is set; then it reads the X/Y position of the cursor and

store it in the X and Y registers. To set the position, the Carry flag must be cleared. We did that with CLC.

Also notice where we used our LDA. We wanted to load the A register *after* the JSR to the PLOT subroutine. The reason for this is that the PLOT subroutine scrambles the A register. It doesn't do this to make life difficult for the programmer, but rather the subroutine itself uses the accumulator. Therefore if we LDA the accumulator with our 88, to get an 'X' printed to our position PLOTted, after the JSR to $FFFØ, we get what we want.

The values for the X register with the PLOT subroutine at $FFFØ can be from Ø-24 and the Y values from Ø-79. Since the screen is made up of 25 rows (Ø-24) the X register values make a lot of sense. However, we know our screen has only 4Ø columns (Ø-39); so what happens when we have Y values from 4Ø-79? I'm not going to tell you. You'll have to change the Y value in the above program to see for yourself!

In addition to using the LDX and LDY instructions in the immediate addressing mode, it is also possible to use them in the absolute mode, just as we did with the LDA instruction. In fact, most of the same addressing modes of the LDA instruction also apply to the LDX and LDY instructions, but there are important exceptions we will see in a bit.

TRANSFERS WITH TAX, TAY, TXA and TYA

Transferring information between the A register (accumulator) and the X and Y registers is handled by single byte instructions. In looking at and remembering how to use TAX, TAY, TXA and TYA, remember you Transfer From-To.

TAX

T from A to X: TAX
T from A to Y: TAY
T from X to A: TXA
T from Y to A: TYA

If we have the value 22 in the accumulator and issue a TAX instruction, the contents of the accumulator are transferred to the X register. Now *both* the A register and the X registers would contain the value 22. None of the Transfer instructions affect the contents of the register from which the contents were transferred. In fact, rather than actually transferring contents, the Transfer instructions *duplicate* the contents in the target register. Therefore, you might want to think of the Transfer instructions as Duplicate instructions.

Now, suppose you want to transfer the contents of the X register into the Y register. There is *no* TXY opcode. If you really think about it, though, I'll bet you can guess how to do it. (Think for a bit before going on. Think. Think. Think.) Okay, time's up. If you guessed you would transfer the X register to the accumulator with TXA and then using TAY, transfer the contents of the accumulator to the Y register, you're absolutely right. Thus, you will often see the following:

```
TXA
TAY
```

Likewise, you can transfer the Y register to the X register using the TYA-TAX sequence. Let's take a look at a program that will show you some work with these registers. See if you can guess what will be printed on your screen before you SYS your program.

GENERAL

```
LABEL          OPCODE   OPERAND COMMENT
-------------------------------------------------------------------------------
{MERLIN        ORG      $C000}
{COMMODORE     * = $C000}

               JSR      $E544
               LDX      #$54
               TXA
               JSR      $E716
               LDX      #$41
               TXA
               STA      49200
```

```
                    LDY     49200
                    TYA
                    JSR     $E716
                    LDY     #$58
                    TYA
                    JSR     $E716
                    RTS
{COMMODORE .END}
```

KIDS' ASSEMBLER

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------
49152   JSR     $E544
49155   LDX#    $54
49157   TXA
49158   JSR     $E716
49161   LDX#    $41
49163   TXA
49164   STA     49200
49167   LDY     49200
49170   TYA
49171   JSR     $E716
49174   LDY#    $58
49176   TYA
49177   JSR     $E716
49180   RTS
```

If you guessed the message is what transfers the contents of the accumulator to the X register, you got it right! We did a lot of unnecessary transferring, but it was more to help you see what's going on in your computer than an example of efficient programming. Let's see what happened.

After clearing the screen, the X register was loaded in the immediate mode with $54 (84 decimal). That value was transferred to the accumulator and then output to the screen. Thus, we got our 'T.'

After that, we loaded the X register with $41 (65 decimal) and transferred it to the A register. Then the value was stored in address 4920Ø with the STA instruction. From the absolute mode we loaded the value in 4920Ø into the Y register, and then transferred it to the A register and printed it to the screen. That's where we got the 'A.'

Finally, the Y register was loaded with $58 (88 decimal), transferred to the accumulator with TYA and the output to the screen. This was how the 'X' was produced.

At this point, it might seem that the transfer instructions do little more than complicate an otherwise simple process; however, as we go on, we will see how they can be very useful in programming. Our above example was just to give you some practice in using them.

INCREMENTING AND DECREMENTING WITH INX, INY, DEX AND DEY.

The incrementing and decrementing of the X and Y registers will play a crucial role in your programming later on. Here, we're only going to see what happens when the INX, INY, DEX or DEY instruction occurs. Basically, when an INX or INY instruction is issued, + 1 is added to the X or Y register. With a DEX or DEY instruction, 1 is subtracted from the X or Y register. By transferring the values of the X and Y registers to the accumulator and sending the results to the screen, we can graphically see what happens.

GENERAL - DEX

```
LABEL          OPCODE   OPERAND COMMENT
-------------------------------------------------------------------------
{MERLIN        ORG      $CØØØ}
{COMMODORE     * = $CØØØ}

               JSR      $E544
               LDX      #9Ø
               LDY      #65
               TXA
```

```
                JSR     $E716     ; Output to
                                    screen
                TYA
                JSR     $E716
                DEX               ; Subtract 1
                                    from X register
                TXA
                JSR     $E716
                INY               ; Add 1 to Y
                                    register
                TYA
                JSR     $E716
                RTS
{COMMODORE      .END}
```

KIDS' ASSEMBLER - DEX

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     JSR     $E544
49155     LDX#    90
49157     LDY#    65
49159     TXA
49160     JSR     $E716
49163     TYA
49164     JSR     $E716
49167     DEX
49168     TXA
49169     JSR     $E716
49172     INY
49173     TYA
49174     JSR     $E716
49177     RTS
```

You should have gotten the following on the upper left-hand corner of your screen:

```
ZAYB
```

The 90 is ASCII value for 'Z' and the 65, the value for 'A'. Since the X register was decremented, the next value would be 89, the ASCII value for the letter 'Y'. Conversely, since the Y register was incremented, it went from 65 to 66 to produce the 'B'.

FROM X AND Y TO MEMORY WITH STX AND STY

Rather than transferring the X and Y register values to the accumulator, and then using STA, it is possible to directly store in memory with STX and STY. The three-byte instructions work in exactly the same way as STA except instead of storing the A register's contents, the X or Y register's contents are stored in the address specified in the operand. Thus, if you program,

```
STY $C100
```

the contents of Y are stored in address $C100.

Let's do something useful with our new knowledge. We'll write another utility program. This one will allow you to link two BASIC files together. As you know, as soon as you LOAD a BASIC program into memory, the current one is replaced by the one just LOADed. This is a real pain in the neck if you have a lot of subroutines you'd like to append to a program you're working on. For example, let's say that you have a handy sort routine stored in a file named SORT. The program you're working on in memory needs that program, but since you can't LOAD it without knocking out your current program, you have key in the whole darned SORT routine from scratch. Wouldn't it be nice if you could just append the SORT routine without having to re-key it in? In fact, wouldn't it be great if you could store all of your handy subroutines in separate files, and just load them up into the file in memory? In that way you could cut your BASIC programming time down considerably.

The next two assembly programs will allow you to LOAD a BASIC file into memory and not destroy the contents of memory. It does this by tricking BASIC into believing that the LOAD address of the second program is the beginning of BASIC RAM. Actually, it loads the program onto the end of the program in

memory. This is done simply by resetting the pointers showing the beginning of the BASIC program to the end of the program in memory. Then, a second machine language program resets the pointers to the actual beginning of BASIC thereby linking the two programs together. The only constraint on these two little utilities is that the second and subsequent programs you LOAD have to have higher line numbers than the ones they're loaded on top of.

GENERAL - APPEND

```
LABEL           OPCODE   OPERAND COMMENT
-------------------------------------------------------------------------------
{MERLIN         ORG      $C000}
{COMMODORE      * = $C000}

                LDX      $2D
                DEX
                DEX
                STX      $2B
                LDX      $2E
                STX      $2C
                RTS
{COMMODORE      .END}
```

GENERAL - LINK

```
LABEL           OPCODE   OPERAND COMMENT
-------------------------------------------------------------------------------
{MERLIN         ORG              $C030}
{COMMODORE      * = $C030}               ; Note address
                                         change

                LDA      #1
                STA      $2B
                LDA      #8
                STA      $2C
                RTS
{COMMODORE      .END}
```

MULTI-LOADER PROGRAM (Merlin)

```
10 IF X = 0 THEN X = 1 : LOAD "APPEND.O",8,1
20 IF X = 1 THEN X = 2 : LOAD "LINK.O",8,1
```

Since the addresses $2B-$2E are zero-page addresses, we need special opcodes on the Kids' Assembler. The "-Z" on the end of an opcode indicates a zero-page operation. Only two bytes are used instead of three. The Merlin and Commodore assemblers automatically recognize zero-page addresses and do not require special indicators.

KIDS' ASSEMBLER - APPEND

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------------------
49152     LDX-Z     $2D
49154     DEX
49155     DEX
49156     STX-Z     $2B
49158     LDX-Z     $2E
49160     STX-Z     $2C
49162     RTS
```

KIDS' ASSEMBLER - LINK

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------------------
49200     LDA# 1                  <- Note start adrs
49202     STA-Z     $2B
49204     LDA#      8
49206     STA-Z     $2C
49208     RTS
```

MULTI-LOADER PROGRAM (Kids' Assembler)

```
10 IF X = 0 THEN X = 1 : LOAD "APPEND 49152",8,1
20 IF X = 1 THEN X = 2 : LOAD "LINK 49200",8,1
```

To see how to use your new utilities, enter the following two BASIC programs, and SAVE each to disk or tape as PART 1 and PART 2:

PART 1

```
10 REM PART 1
20 REM LOAD FIRST
```

PART 2

```
30 REM PART 2
40 REM APPENDS TO PART 1
```

After you have a copy of both programs SAVEd, load your two machine files, APPEND and LINK, with the Multi-loader program. Next, LOAD "PART 1" and LIST the program. There should be just two lines, 10 and 20. Now, SYS 49152. This will activate APPEND, and you can now LOAD "PART 2". When you LIST the program again, you will see line 10-40 all together as a single program. When you SYS 49200, the pointers will be reset, and the two programs will be treated as a single large program.

The nice thing about machine language utility programs is that they will stay in memory while to work with your BASIC files. You can even RUN your BASIC programs without hurting them. In that way, you can load the utilities at the beginning a programming session and forget about them until they're needed to append and link BASIC files.

ADDRESSING MODES WITH THE X AND Y REGISTERS

In the next chapter, we will see how important the X and Y registers are in loops and branches. In fact, that is where their real utility lies, saving you a lot of time in programming. At this point, we will simply show you what the various addressing modes using the X and Y registers do.

We have already seen that the immediate and absolute modes with LDX and LDY work the same as with LDA. However, we can

also do indexed addressing using the X and Y registers as offsets to our intended address.

INDEXED ABSOLUTE ADDRESSING

Basically, the way indexed addressing works with your programs is to *add* the value of the X or Y register to the address in the operand. For example, look at the following program:

GENERAL

```
LABEL        OPCODE  OPERAND  COMMENT
-------------------------------------------------------------
             LDX #Ø
             TXA
             STA     $4ØØ,X   ; Stores at $4ØØ
             INX              ; Add 1 to X
             TXA
             STA     $4ØØ,X   ; Store at $4Ø1
```

KIDS' ASSEMBLER

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------
49152    LDX#    Ø
         TXA
         STA-X   $4ØØ
         INX
         TXA
         STA-X   $4ØØ
```

The program simply loads the X register with Ø, transfers the Ø to the accumulator and then stores the Ø in the specified address ($4ØØ) plus the value of the X register. So the first value is stored at $4ØØ + Ø, or $4ØØ. Then the X register is incremented by 1; so now the value of X is 1 (Ø + 1 = 1). That is transferred to the accumulator and stored in $4ØØ + X. Since X is now 1, $4ØØ + 1 = $4Ø1. Therefore, even though the address in the operand is still indicated as $4ØØ, it is actually $4ØØ + contents of the X register as far as the computer is concerned. The following is the general formula of what occurs with indexed absolute addressing:

Indexed Absolute Addressing

ADDRESS = ADDRESS + VALUE OF X OR Y REGISTER

To see how we can put this to use, let's pick a location that we can see incremented. We know the screen addresses 1024-2023 ($400-7E7) make up our screen memory and 55296-56295 ($D800-DBE7) make up the color memory map. By using indexed addressing, we can place values in those locations simply by incrementing X or Y and storing our values in the starting addresses offset by the X or Y registers.

GENERAL - INDEXED ADDRESSING

```
LABEL           OPCODE   OPERAND COMMENT
-------------------------------------------------------------------------
{Merlin         ORG              $C000}
{Commodore      * = $C000}

                JSR      $E544
                LDA      #4
                STA      $D021    ; Background
                                  color
                LDA      #1
                LDX      #0
                STA      $D800,X  ;Base color
                                  adrs + X
                STA      $400,X   ;Base screen
                                  adrs + X
                INX
                STA      $D800,X
                STA      $400,X
                INX
                STA      D800,X
                STA      $400,X
                RTS
{Commodore:     .END}
```

KIDS' ASSEMBLER - INDEXED ADDRESSING

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------------------
49152     JSR       $E544
          LDA#      4
          STA       $D021
          LDA#      1
          LDX#      0
          STA-X     $D800
          STA-X     $400
          INX
          STA-X     $D800
          STA-X     $400
          INX
          STA-X     $D800
          STA-X     $400
          RTS
```

The program clears the screen and turns the background color purple. The value '1' is loaded into the accumulator and '0' into the X register. We will not be changing the value in the accumulator, but simply storing it in the screen and color memory areas so that we can see what's happening. The first storage location is at the base addresses of $D800 and $400. The X register is incremented, and the second STA is in $D801 and $401. The X register is incremented once again, providing indexed addresses of $D802 and $402. Your screen results should be three white A's in the upper left hand corner.

With our current set of instructions, we could have provided the actual addresses ourselves and used a few less instructions. However, using indexed addressing, it is a little more convenient to use the X register to increment the base addresses. When we get to loops, you will really see the use of indexed addressing. Finally, we could have used the Y register as our index in exactly the same way.

The next two addressing modes are a little hairy and will be introduced here just to show you how they work. They will be more

useful as you become more advanced. You might want to skip this last section and go on to the next chapter.

INDEXED INDIRECT ADDRESSING (X Register Only)

If you ever played pool, you probably know what a "bank shot" is. Instead of making your shot to send the ball directly into a pocket, you bounce it off the side first. Indexed indirect addressing is a type of "bank shot" used to check, set or obtain a value. The following illustrates the basic concept:

ADDRESS = POINTER STORED IN ZERO PAGE + X REGISTER VALUE = LOCATION OF ADDRESS

In zero page ($Ø-FF) a set of pointers are stored in 2 byte configurations. The low-byte is at the lower address and the high-byte is at the higher address. For example, let's say you have pointers stored in $FB-$FE (251-254). They have the following contents:

Indexed Indirect Shot

$FB-$FC $AB $CØ
$FD-$FE $DØ $CØ

The addresses are stored low-byte / high byte, therefore we are actually looking at $CØAB and $CØDØ. Note that the two addresses are not sequential. They do not have to be, but the pointers are sequential. Therefore, since each pointer takes up two bytes, the indexing will have to be in steps of two.

When indexed indirect addressing is used, you either fetch a value from the pointed-to address or store a value there. For example, let's say that $CØAB has the value 1Ø ($ØA) and $CØDØ has 2Ø($14). Using the format,

LDA ($FB,X) ◄-General

or

(LDA-X) $FB ◄-Kids' Assembler

where X is the contents of the X register, you would load either the contents of $CØAB or $CØDØ. Let's say the X register is 'Ø'. The following would occur:

```
LDA ($FB,X)
$FB + Ø = $FB ◄- Get the address from $FB and $FC
$FB = $AB $FC = $CØ
Address to get value = $CØAB
Contents of $CØAB = 1Ø
Accumulator is loaded with 1Ø
```

If the X register is incremented by 1, and indexed indirect addressing is used, you'd be in trouble. That's because the pointer would show $FB + 1 to be the beginning of the two byte address. Thus, your load would be from,

```
$FC = $CØ
$FD = $DØ
```

which points to $DØCØ. That address is in the I/O, Color RAM or Character Generator area where you'd probably not be storing variables. Instead, you'd want to be sure you indexed by 2. Supposing the value of the X register is 2, you'd get the following:

```
LDX #2
LDA ($FB,X)
$FB + 2 = $FD ◄- Get the address from $FD and $FE
$FD = $DØ $FE = $CØ
Address to get value = $CØDØ
Contents of $CØDØ = 2Ø
Accumulator is loaded with 2Ø
```

One of the reasons we won't be using this addressing mode much in this book is because it relies on a Zero-page address for its pointers. Since the Kids' Assembler is written in BASIC and all kinds of BASIC pointers are stored in zero-page, we'd be very limited in the areas we could use. Also, since we will want to use our machine language subroutines interactively with BASIC, we could easily bomb the BASIC program we're working with if we stored several pointers in zero-page. However, to give you a simple example of this type of addressing mode, try the following example:

GENERAL INDEXED INDIRECT ADDRESSING

```
LABEL           OPCODE  OPERAND     COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}

                JSR     $E544
                LDY     #65         ;Load Y with
                                    ASCII 'A'
                STY     $C100       ;Store Y in
                                    $C100
                LDA     #$00        ;Low byte of
                                    target adrs.
                STA     $FB         ;Store in LB
                                    pointer adrs.
                LDA     #$C1        ;High byte of
                                    target adrs.
                STA     $FC         ;Store in HB
                                    pointer adrs.
                LDX     #$0
                LDA     ($FB,X)     ;Indexed in-
                                    direct LDA
                JSR     $E716       ;Output to
                                    screen
                RTS
{Commodore      .END}
```

= = ZERO-PAGE ADDRESSING = =

Zero page addressing has special machine language opcodes that are not apparent in most assemblers. The STA instruction in the absolute mode has one opcode for non-zero page addressing and another for zero page addressing. Thus, STA instructions for $1ØØ and higher are interpreted as machine opcode $8D, but for locations of $ØFF and lower, the machine opcode $85 is used. Zero-page addressing saves one byte compared with non-zero page addressing since the operand address is one byte ($FF or less.) Since the Kids' Assembler has a one-to-one correspondence between assembler opcode and machine opcode, it is necessary to have special opcodes indicating a zero-page operation, indicated by '-Z' extenders. This method was used on the Append and Link programs in this chapter already.

KIDS' ASSEMBLER - INDEXED INDIRECT ADDRESSING

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------------------
49152     JSR       $E544
          LDY#      65
          STY       $C1ØØ
          LDA#      $ØØ
          STA-Z     $FB
          LDA#      $C1
          STA-Z     $FC
          LDX#      $Ø
          (LDA-X)   $FB
          JSR       $E716
          RTS
```

The above program is a pretty weird way to get a crummy 'A' printed to the screen. However, it shows what must be done to set up indexed indirect addressing. First, the target address must be

given a value to use. Second, the low and high byte of the target address must be stored in zero-page. Finally, the X Register must be given a value. After that is done, indexed indirect addressing is possible. When you begin using tables of numbers, you can use the X Register as an indirect pointer to the table. In the meantime, I wouldn't spend a lot of time trying to use this mode.

INDIRECT INDEXED ADDRESSING (Y Register Only)

The final addressing mode we will discuss in this chapter is indirect indexed. Like the indexed indirect, indirect indexed addressing uses a zero-page pointer. However, instead of using the X register, it uses the Y. Also, instead of pointing to a series of addresses in zero-page, it points to a *single* address offset by the value of Y. This addressing mode is much easier to use since it involves only two bytes of zero-page. The following outlines the mode:

ADDRESS = POINTER STORED IN ZERO PAGE +
X REGISTER VALUE = LOCATION OF ADDRESS

For example, let's say you have pointers stored in $FB-$FC (251-252). They have the following contents:

```
$FB-$FC $D1 $C0
```

Stored in low-byte / high-byte configuration, the base target address is $C0D1. In the indirect indexed mode, the actual address would be $C0D1 + Y Register. Therefore, if the contents of the Y register were 4, the target address would be $C0D1 + 4 or $C0D5. For a table using sequential addresses, this method is useful for accessing those addresses using Y as an offset. The format is,

```
LDA ($FB),Y ◄- General
(LDA-Y) $FB ◄- Kids' Assembler
```

Let's take a look at an example that will take values from three consecutive two-byte addresses and print them to the screen. When you execute the program, 'ABC' will appear in the upper left hand corner of your screen.

GENERAL INDIRECT INDEXED ADDRESSING

```
LABEL          OPCODE  OPERAND    COMMENT
{Merlin        ORG     $C000}
{Commodore     * = $C000}

               JSR     $E544
               LDX     #$D1       ;Low byte target
                                  adrs.
               STX     $FB        ;Low byte
                                  pointer
               LDX     #$C0       ;High byte
                                  target adrs.
               STX     $FC        ;High byte
                                  pointer
               LDX     #65        ;ASCII 'A'
               STX     $C0D1      ;Store in first
                                  target adrs.
               INX
               STX     $C0D3
               INX
               STX     $C0D5
               LDY     #$0        ;Set Y to $0
               LDA     ($FB),Y
               JSR     $E716
               INY
               INY
               LDA     ($FB),Y
               JSR     $E716
               INY
               INY
               LDA     ($FB),Y
               JSR     $E716
               RTS
{Commodore     .END}
```

KIDS' ASSEMBLER - INDIRECT INDEXED ADDRESSING

```
ADRS      OPCODE    OPERAND
-------------------------------------------------------------------------------
49152     JSR       $E544
49155     LDX#      $D1
49157     STX-Z     $FB
49159     LDX#      $C0
49161     STX-Z     $FC
49163     LDX#      65
49165     STX       $C0D1
49168     INX
49169     STX       $C0D3
49172     INX       $C0D3
49173     STX       $C0D5
49176     LDY#      $0
49178     (LDA-Y)   $FB
49180     JSR       $E716
49183     INY       $FB
49184     INY       $E716
49185     (LDA-Y)   $FB
49187     JSR       $E716
49190     INY
49191     INY
49192     (LDA-Y)   $FB
49194     JSR       $E716
49197     RTS
```

As we pointed out, you won't be using the indexed indirect and indirect indexed modes much at first. However, one of the best ways to learn assembly language programs is to look at other people's work. By knowing about these two modes of addressing, you'll be better able to understand what others are doing. In the meantime, though, don't worry about them. You won't need them much at the beginning level.

CHAPTER 10

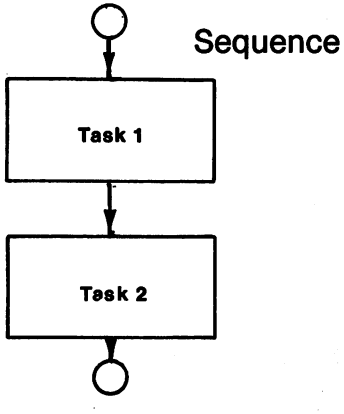
LOOPS AND BRANCHES

PROGRAM STRUCTURES

At the first mention of program structure, a lot of people groan that it's enough just to get the program done and working without having to worry about "structured programming." That's true, and I certainly won't lecture on the merits of structured programming except insofar as it makes life easier. All I want to do here is to explain the fundamental structures in all programming. There are only three!

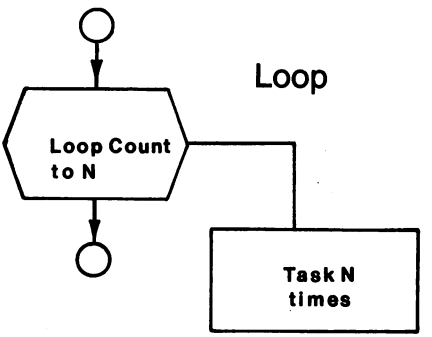
SEQUENTIAL

So far, all we have dealt with in assembly programs is the sequential arrangement of instructions. Each instruction is executed one-after-the-other. It's like climbing a ladder one rung at a time going in the same direction.



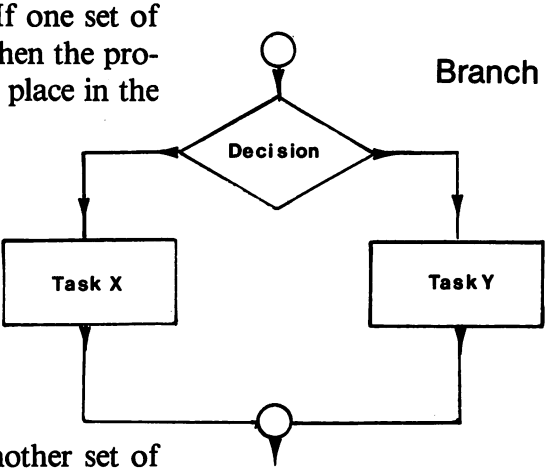
LOOPS

A loop is the second structure. At a certain point in the program, the program goes back to an earlier part and does it all over again. Most loops have "escape clauses." That means that after executing the loop a given number of times or meeting some other condition, such as reading a keypress, it exits the loop and goes on to the next sequential instruction. You're familiar with the FOR/NEXT loop in BASIC.



BRANCHES

A branch occurs when the program can take two or more courses of action. If one set of conditions is met, then the program jumps to one place in the program, and if another set of conditions is met; then it jumps to another. The IF/THEN statement and GOTO and GOSUB statements in BASIC are example of branch structures.



That's all there is to structure in programming. (Well, almost.) It's painless, and as we will see, very important to make your job of programming easier.

HOW BASIC LOGIC WORKS IN ASSEMBLY LANGUAGE

Remember that the fundamental difference between BASIC and assembly language programming is that you have to provide more information in assembly language than you do in BASIC. Otherwise, both use the same structures and logic. We've already seen how sequential structure works in both types of programming. For example, to clear the screen and print the letter 'A' to the screen, the following two structures are used:

Sequence

BASIC

```
10 PRINT CHR$(147)
20 PRINT "A"
```

Assembly

```
JSR  $E544
LDA  #65
JSR  $E716
RTS
```

There's really not a lot of difference in the amount of work you have to do for either. However, to print 'ABC' to the screen, there's a big difference in the effort involved in sequential programming:

BASIC

```
10 PRINT CHR$(147)
20 PRINT "ABC"
```

Assembly

```
JSR  $E544
LDA  #65
JSR  $E716
LDA  #66
JSR  $E716
LDA  #67
JSR  $E716
RTS
```

With math programs there is even a bigger difference in the amount of work involved, but we'll get to that in the next chapter. For now, it is enough to see that sequential structures in BASIC take a lot less programming than in assembly language programming.

LOOPING TO SAVE PROGRAMMING TIME

Let's say you wanted to print the alphabet to the screen. In BASIC you could do something like the following:

```
10 PRINT "A"
20 PRINT "B"
30 PRINT "C"
```

etc.

However, you'd probably use a loop and enter something like the following:

```
10 FOR X = 65 TO 90 : PRINT CHR$(X); : NEXT X
```

You use the loop structure because it takes a lot less time to program and does the same thing as separate PRINT statements. In other words, it's a *smarter* way to structure your program.

Now, if you can save steps in BASIC using loops, just think of what you can save in assembly programming! Remember, since you have to give a lot more instructions to get something done in

assembly language, if you could use those same instructions in a loop, you could do a lot more with less code.

The question is, "How?" The best way to think of a loop in assembly language programming is as a "branch back." In BASIC when your program encounters a NEXT statement, it is really telling the program to "branch back" to the FOR statement. Secondly, there must be some kind of comparison or test to exit the loop; otherwise you'd be stuck in an endless loop. With the BASIC FOR/NEXT statement, the test is the top of the loop. (i.e., The last number in the FOR statement.) In summary, the loop would look as follows:

1. BEGIN LOOP
2. COMPARE FOR MATCH OR NO MATCH
3. IF NO MATCH GO BACK TO BEGINNING OF LOOP
4. OUT OF LOOP

Let's look at that in a BASIC program using a GOTO loop and IF/THEN comparison to print the alphabet instead of the FOR/NEXT statements:

```
10 X = 65 : REM INITIALIZE X
20 PRINT CHR$(X) : REM BEGIN LOOP
30 X = X + 1 : REM INCREMENT X
40 IF X < > 90 THEN GOTO 20 : REM COMPARE AND
LOOP
50 PRINT "ALL DONE"
```

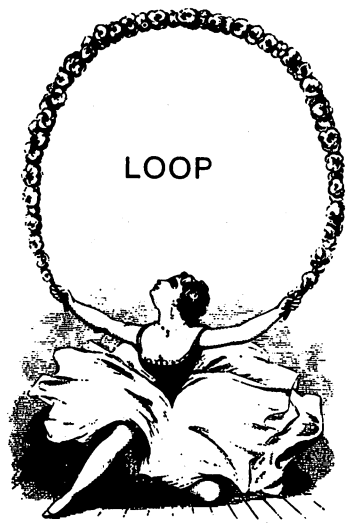
Now all we need are opcodes to Compare and Branch. For comparison, we will use the following three:

Compare OPCODES

CMP	Compare value with accumulator
CPX	Compare value with X register
CPY	Compare value with Y register

Next, we need Branch opcodes.

BNE Branch not equal – <>
BEQ Branch if equal – =



Now, let's take a look at the equivalent assembly program to the last BASIC one:

GENERAL - ALPHABET LOOP

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore     * = $C000}

               JSR     $E544
               LDX     #65      ;Initialize X
                                with 65 - ASCII
                                A
LOOP           TXA
               JSR     $E716
               INX              ;Increment X
               CPX     #91      ; Compare X
                                with 91
               BNE     LOOP     ; If X <> 91
                                then branch
                                back to LOOP
               RTS
```

KIDS' ASSEMBLER - ALPHABET LOOP

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------------------------
49152    JSR     $E544
49155    LDX#    65
49157    TXA
49158    JSR     $E716
49161    INX
49162    CPX#    91
49164    BNE     49157
49166    RTS
```

Now the first thing to notice in the programs is the different ways in which the branch was used. Generally, assemblers will have provisions for a LABEL field. In that field you can label your lines, with the label serving as an address. In the Kids' Assembler, since it has no label field, you have provide the address to the branch. (That's one of the reasons the Kids' Assembler gives you the addresses as you program.)

= = A BRANCH TOO FAR = =

At this stage of the game, you won't be likely to try a branch that leaps more than a few addresses. If you do branch more than 128 bytes backwards or 127 forward, you'll get in trouble. All branch instructions are coded as branch offsets between Ø-255 ($Ø-$FF) in machine code and not as addresses. That's why the branch instructions are only two bytes instead of three. There is a trick to branching further forward or backwards than 127 or – 128 by inserting a JMP instruction within the range of your branch. However, you won't need it for the programs in this book.

At the beginning of this chapter we looked at a program in assembly language that printed 'ABC' to the screen. It took eight lines of code since we had to keep putting new values in the ac-

cumulator with LDA and then jumping to the screen output routine. In the last program, we printed all 26 letters of the alphabet also using eight lines. In other words, we were able to do almost nine times the output with the same amount of code. As you can see, using the loop structure saved us a lot of time.

INDEXING WITH LOOPS

In the Chapter 9, we examined indexed addressing with the X and Y registers. Using indexed addressing we saved a little programming time since we could increment our base address with X or Y and didn't have to figure it out every time we wanted to use the next address. With loops, however, we can really do a lot with indexed addressing. Take a look at the following assembly program to see how we can send information to the sequential addresses of the screen very easily.

```
LABEL           OPCODE  OPERANDCOMMENT
-------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}

                JSR     $E544
                LDX     #0
                LDY     #1
START           TYA
                STA     55296,X
                TXA
                STA     1024,X
                INX
                CPX     #255
                BNE     START
                RTS
{Commodore      .END}
```

KIDS' ASSEMBLER

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------------------------
49152    JSR     $E544
49155    LDX#    0
49158    LDY#    1
49161    TYA
49162    STA-X   55296
49165    TXA
49166    STA-X   1024
49169    INX
49170    CPX#    255
49173    BNE     49161
49175    RTS
```

The loop allowed us to use the base address for the character and color screen, and indexing by X, store all 255 characters to the screen. Using the Y value, we did the same thing with the corresponding color addresses on the screen.

The next programs we will examine, show us two things. First, after running the BASIC version of the program, you will see the incredible speed of a machine language program that does exactly the same thing. Secondly, you will see how we can use several address bases within a loop to speed things up. Since the A,X and Y registers can only hold 255, by having offsets from the base addresses, we can put these offsets in our program to access more than the base address + 255, indexed by the X register. (Be sure to run the BASIC program first, so that you can see what is happening on the screen.)

BASIC - FILL SCREEN

```
10 PRINT CHR$(147)
20 FOR X=0 TO 249
30 POKE 55296+X,X : POKE 55546+X,X
40 POKE 55796+X,X : POKE 56046+X,X
60 POKE 1024+X,X : POKE 1274+X,X
70 POKE 1524+X,X : POKE 1774+X,X
80 NEXT
```

GENERAL - FILL SCREEN

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore     * = $C000}

               JSR     $E544
               LDX     #0
LOOP           TXA
               STA     55296,X    ;Base color
                                  addrs.
               STA     55546,X
               STA     55796,X
               STA     56046,X
               STA     1024,X     ;Base screen
                                  addrs.
               STA     1274,X
               STA     1524,X
               STA     1774,X
               INX
               CPX     #250       ;Compare X
                                  value with 250
               BNE     LOOP
               RTS
{Commodore     .END
```

KIDS' ASSEMBLER - FILL SCREEN

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     JSR     $E544
49155     LDX#    0
49157     TXA
49158     STA-X   55296
49161     STA-X   55546
49164     STA-X   55796
49167     STA-X   56046
49170     STA-X   1024
49173     STA-X   1274
```

```
49176    STA-X    1524
49179    STA-X    1773
49182    INX
49183    CPX#     250
49185    BNE      49157
49187    RTS
```

We used the value 250 for our test so that we could use four equal blocks to fill the 1000 addresses of our screen. Also notice that we used the same values for our color codes as the character codes.

NESTED LOOPS

In addition to using loops in assembly language just as you can in BASIC, so too can you use nested loops. Like regular loops, nested loops can save you a lot of programming time. For example, in our last program, we had to put four base addresses as offsets for the indexed addressing we were using to fill up the screen. The reason we had to do that was because of the 255 ($FF) limitation in the X register. If the X register could hold 1000, we could have been able to use a single base address for the character and color bases.

Using a slightly different example, we will now look at a way to fill up your screen with printable characters using nested loops. Since the JSR $E716 routine outputs to screen in sequential order, as long as we JSR $E716 1000 times, we can fill the screen. Now we know that the X and Y registers are limited to 256 (0-255) before they burp and fall over, but by using *both* registers and nested looping, we can get up to 256 x 256 ($FFFF) passes through a loop. Let's start with a BASIC program so we can leisurely watch it and then do the same with an assembled machine language program. Notice how in all cases, the second loop is *inside* the first loop. (We only used ASCII 33-127 to output since the others are color changers and screen scramblers.)

BASIC - NESTED LOOP

```
10 PRINT CHR$(147)
20 FOR Y = 0 TO 9 : REM LOOP1
30 FOR X = 33 TO 127 : REM LOOP2
40 PRINT CHR$(X);
50 NEXT X : REM BRANCH TO LOOP2
60 NEXT Y : REM BRANCH TO LOOP1
```

GENERAL - NESTED LOOP

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore     * = $C000}

               JSR     $E544
               LDY     #0
LOOP1          LDX     #33
LOOP2          TXA
               JSR     $E716
               INX
               CPX     #127
               BNE     LOOP2
               INY
               CPY     #10
               BNE     LOOP1
               RTS
```

KIDS' ASSEMBLER - NESTED LOOP

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     JSR     $E544
49155     LDY#    0
49157     LDX#    33
49159     TXA
49160     JSR     $E716
49163     INX
```

```
49164    CPX#    127
49166    BNE     49159
49168    INY
49169    CPY#    10
49171    BNE     49157
49173    RTS
```

You may have noticed the the program was a little slower than our last one. That was because the JSR $E716 takes more time than a simple STA. The JSR involves executing a subroutine and returning, while the STA is a single machine language instruction.

BRANCHING FORWARD WITH JMP, BEQ AND BNE

As we have seen with our loops, the branching that occurs is all backwards. You might ask, "Why jump forward? Why not complete a task, either with or without a loop and then go on to the next part of the program?" The essence of good programming is what's called a "Top Down" structure. You begin at the beginning and proceed in an orderly, sequential fashion to the end of the program. Jumping all over the place is called "spaghetti" programming. Essentially, structured programming looks like the following:

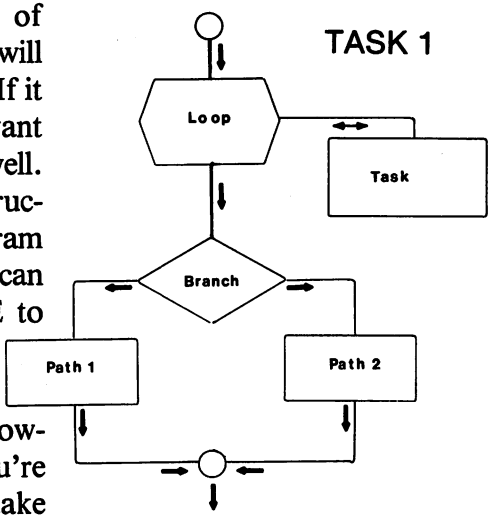
```
TASK 1

 ◄CHECK► ALL DONE? (YES/NO)
 – NO: GO BACK AND FINISH
 – YES: GO ON TO NEXT TASK

TASK 2
 ◄CHECK► ALL DONE? (YES/NO)
 – NO: GO BACK AND FINISH
 – YES: GO ON TO NEXT TASK
.....
TASK END
 ◄CHECK► ALL DONE? (YES/NO)
 – NO: GO BACK AND FINISH
 – YES: EXIT PROGRAM
```

Unstructured program

Using structured programming techniques, you can save a lot of time and have your programs run better. However, treat structured programming techniques as tools and not religion! The Top-Down structure is important, but there are exceptions to its strict interpretation. Every time you JSR to a subroutine, you leave the sequential path, and there will be instances where a task is accomplished in an unstructured fashion. Just remember though, the more structured your program the easier it is to write, run and debug.

The following situation is a common one in assembly language programming:

In TASK 1, your program loops until the exit condition is met. Depending on the outcome of the loop, your program will branch to Path 1 or Path 2. If it goes to Path 1, you do not want it to go through Path 2 as well. This is where the JMP instruction comes in. If your program is to branch to Path 2, you can simply JMP, BEQ or BNE to the beginning of Path 2. However, if you take Path 1, you're probably going to have to take an unconditional jump using the JMP instruction. Look at the following program:



In TASK 1, your program loops until the exit condition is met. Depending on the outcome of the loop, your program will branch to Path 1 or Path 2. If it goes to Path 1, you do not want it to go through Path 2 as well. This is where the JMP instruction comes in. If your program is to branch to Path 2, you can simply JMP, BEQ or BNE to the beginning of Path 2. However, if you take Path 1, you're probably going to have to take an unconditional jump using the JMP instruction. Look at the following program:

```
LABEL       OPCODE  OPERAND  COMMENT
----------------------------------------------------------------------
            LDX     #0
LOOP        INX
            CPX     $C200    ;Keep looping
                             until X= con-
                             tents of $C200
            BNE     LOOP
            CPX     $8000    ;After loop is
                             X= contents of
                             $8000?
            BEQ     PATH2    ;If equal then
                             PATH2
PATH1       INX
            TXA
            JSR     $E716
            JMP     END      ;Jump over
                             PATH2 to end
PATH2       DEX
            TXA
            JSR     $E716
END         RTS
```

In the above program, locations $C200 and $8000 have variable values stored. The X register is incremented until it is equal to the contents of $C200. When X equals that amount, X is compared with the contents of $8000. If they're equal, the program takes PATH2; otherwise it takes PATH1. At the end of PATH1, it JuMPs over PATH2 to the END of the program. (The above program is hypothetical to illustrate a point. If you want to run it, provide values for $C200 and $8000.)

In Chapter 12 when we discuss interacting with a program from the keyboard, you'll be using JMP and BEQ more frequently. In the meantime, let's look at a little program that uses both BEQ and JMP.

GENERAL - BEQ AND JMP

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore     * = $C000}

               LDX     #65
START          TXA
               JSR     $E716
               CMP     #90
               BEQ     END
               INX
               JMP     START
END            RTS
```

KIDS' ASSEMBLER

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     LDX#    65
49154     TXA
49155     JSR     $E716
49158     CMP#    90
```

All the program does is print the alphabet to your screen by incrementing X until it reaches the value 90. We could have done the same thing with BNE and used less code, but it did allow us to place the INX after the comparison and branch instructions. The main point of the program was to illustrate how BEQ and JMP work.

SUMMARY

The main point of this chapter was to show you ways to make assembly language programming easier. Just as in BASIC where you can cut down considerably on the amount of work you do by using loop structures, loops in assembly language save you a good deal of time as well. Perhaps the most important thing to remember is using loops with X and Y indexed addressing. The base address of a block of RAM can be accesses with a minimal amount of program code.

Overall, you can think of the three structures in programming, sequential, loop and branch, in the same way you would in BASIC. In solving a programming problem, use the same structural tools you would in BASIC. The substitution is in the instructions you give; not the fundamental logic of those structures. In the next chapter, we will take a look at some additional instructions that can be used to structure your program and save you time.

CHAPTER 11

ADDING AND SUBTRACTING

INCREMENTING AND DECREMENTING MEMORY : INC AND DEC

We saw that we could increase or decrease the value of the X and Y registers with INX, INY, DEX and DEY. By using the X and Y registers, we could indirectly increase or decrease the value in memory. For example, the following program would increment the contents of location $C1ØØ using the X register:

```
LABEL    OPCODE  OPERAND
-------------------------------------------------------------------------------
         LDX     #$Ø
START    STX     $C1ØØ
         INX
         CPX     #$FF
         BNE     START
         RTS
```

That method works fine, and the same could be done with the Y register and decrementing values as well. However, there will be occasions when you will want to use the X or Y registers for something else while incrementing or decrementing values. For example, suppose you want to use the X register for an inside loop, the Y register

for an outside loop, incrementing both registers but you wanted to decrement the values in a series of addresses.

The INC instruction simply increments the value of the target address by one in the absolute mode. For example, if location $C100 contained the value 5,

```
INC     $C100
```

would increment the value in $C100 to 6. Similarly, if DEC had been used, the value would be decreased by 1 to 4.

To see how INC works, we'll write a program that will run through the alphabet for us. (Nothing new, but it shows us what's happening. Wait'll we get to graphics and use these instructions!)

GENERAL - ABSOLUTE INC

```
LABEL           OPCODE  OPERANDCOMMENT
-------------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}

                LDA     #64
                STA     $C100
START           INC     $C100
                LDA     $C100
                JSR     $E716
                CMP     #90
                BNE     START
                RTS
{Commodore      .END}
```

KIDS' ASSEMBLER - ABSOLUTE INC

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   LDA#    64
49154   STA     $C100
49157   INC     $C100
```

```
49160    LDA     $C100
49163    JSR     $E716
49166    CMP#    90
49168    BNE     49157
4917Ø    RTS
```

In the above example, we used INC in the absolute mode. Both INC and DEC opcodes can be used in the indexed addressing mode as well. For example, if we changed the above program, we could INCrement a whole series of addresses with $C1ØØ, as the base.

Fill with **INC**

GENERAL - INDEXED INC

```
LABEL          OPCODE  OPERANDCOMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore     * = $C000}
               LDX     #0
               LDA     #64
               STA     $C100
START          INC     $C100,X
               LDA     $C100,X
               JSR     $E716
               INX
               CPX     #90
               BNE     START
               RTS
{Commodore     .END}
```

KIDS' ASSEMBLER - INDEXED INC

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     LDX#    0
49154     LDA#    64
49156     STA     $C100
```

```
49159     INC – X    $C100
49162     LDA – X    $C100
49165     JSR        $E716
49168     INX
49169     CPX#       90
49171     BNE        49159
49173     RTS
```

The results of this program may surprise you. At first, you may have thought you'd get the alphabet printed to the screen and then stored in sequential addresses from $C100-$C11A (49408-49434). That would make a nice table, but that's not what we got. Instead, as you can see from the mess on your screen, after getting the 'A' as expected, the rest is garbage. The reason for that lies in the fact that we were changing the addresses and INCrementing each new address by 1. Thus, while we stored the value 64 in $C100 and then incremented it by 1 to get 65 (the 'A' you saw on your screen), we then INCremented $C101, which was empty (or full of junk) by 1. That junk was sent to the accumulator and output to the screen. We'll need some new instructions to get what we need. (I'll bet you're smart enough to figure out how to get the alphabet stored in a table without any new instructions, though. Have the Y register help you out.)

ADDING AND SUBTRACTING IN THE ACCUMULATOR : ADC AND SBC

The ADC and SBC allow you to ADd to the accumulator with Carry or SuBtract from the accumulator with Carry. Instead of incrementing or decrementing with transfers from the X or Y registers or memory, you can add and subtract as much as you want in just about all modes. The statement,

```
ADC       #09
```

adds +9 what whatever is in the A register. In the absolute mode,

```
ADC       $C100
```

adds the contents of location $C1ØØ to your accumulator. For example, the following program will print 'AZ' to your screen by adding 25 to the accumulator which already contains 65:

GENERAL - INDEXED ADC

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $CØØØ}
{Commodore     * = $CØØØ}
               LDA     #65
               JSR     $E716
               ADC     #25
               JSR     $E716
               RTS
{Commodore     .END}
```

KIDS' ASSEMBLER - INDEXED INC

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------------------------
49152    LDA#    65
49154    JSR     $E716
49157    ADC#    25
49159    JSR     $E716
49162    RTS
```

USING CLC AND SEC

With a single ADC instruction, we can make giant leaps instead of relying on a series of INX's, INY's or INC's. However, when we start to use a series of ADC's, it's necessary to clear the carry flag. (In fact it's a good idea whenever ADC is used.) To do that, you simply use the sequence shown in the following example:

```
CLC
ADC     #$D3
```

The reason for clearing the carry flag (the C flag - remember NV SDIZC?) is because the carry flag may have been set by some other operation. When you ADd with Carry, you add the intended number along with the Carry if it is set. For example, a JSR to an output routine may set the carry flag. In the following program where the accumulator is incremented by *two* each time the program loops, the CLC instruction makes sure that the contents of the accumulator are incremented only by two and not two plus the carry. If you want to see the results without CLC, delete the CLC instruction and place the START label in the ADC line.

GENERAL - ADC BY TWO'S

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}
; ***********************************************
; *                                             *
; *              ADC BY TWO'S                   *
; *                                             *
; ***********************************************

                JSR     $E544
                LDA     #63
START           CLC
                ADC     #2
                JSR     $E716
```

```
                CMP     #89
                BNE     START
                RTS
{Commodore      .END}
```

KIDS' ASSEMBLER - ADC BY TWO'S

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDA#    63
49157   CLC
49158   ADC#    2
49160   JSR     $E716
49163   CMP#    89
49165   BNE     49157
49167   RTS
```

= = GETTING FANCY = =

We used a box of stars (asterisks) in our general assembler program to show you how to give your source code a hot-shot header. On most assemblers, if the first character is a semi-colon, the entire line is read as a comment. It's sort of like having REM statements use entire lines for headers in BASIC programs. On the Merlin Apprentice assembler, you don't even need the semi-colon. If your first entry is CTRL-P, you'll get a line of stars. If you hit the space bar and press CTRL-P the first and last spaces will get stars. It helps to have headers so that you can quickly see what the source code is for. This is especially true if you're working with a lot of different assembly language programs. It is also a good idea to put the date in the header so that you will know when you last worked on the program. With larger programs, you may make several versions, and the dates will keep you posted on which version you have in the editor.

Now, this next part is weird; so put down your root beer and listen up. When we want to ADC, it's important to clear the carry flag with CLC. However, when we subtract from the accumulator with SBC we *want* the carry flag set; so we use SEC to set the carry flag. The reason for this is that in subtraction, the set carry flag is treated as though no borrow is taken; just the reverse of ADC. What you're really doing is called "two's compliment" addition since that's the way your 651Ø can best handle subtraction. It sort of uses the carry flag to "add backwards." Rather than losing sleep over the process, just remember:

SUBTRACT - SEC

Use the format in the following example:

```
SEC
SBC   #$Ø8
```

Okay, we're all set to use subtract. We'll go backwards in the alphabet printing only every third character. In that way you can see how the process works.

GENERAL - SBC BY THREE'S

```
LABEL          OPCODE  OPERAND COMMENT
------------------------------------------------------------------------------
{Merlin        ORG             $CØØØ}
{Commodore     * = $CØØØ}
; ********************************************
;  *                                         *
;  *          SBC BY THREE'S                 *
;  *                                         *
;

               JSR     $E544
               LDA     #93
START          SEC                ;Set the Carry
               SBC     #3         ;Subtract 3 from
                                  the accumulator
               JSR     $E716
               CMP     #66
```

```
                BNE     START
                RTS
{Commodore      .END}
```

KIDS' ASSEMBLER - BY THREES

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDA#    93
49157   SEC
49158   SBC#    3
49160   MSR     $E716
49163   CMP#    66
49165   BNE     49157
49167   RTS
```

SUMMARY

This has been a short chapter, but we learned some important new instructions for adding and subtracting numbers. We did not tackle the more difficult problems of dealing with numbers larger than 255 ($FF) since that gets a little hairy. If you want to add 2 + 2, stick with BASIC for the time being, and the same is even truer when dealing with multiplication and division. Larger numbers can be handled in assembly language using two addresses, and when you become more advanced, you'll learn either how to use JSR's to BASIC subroutines or how to write your own arithmetic programs. As you will see, we'll have our hands full just dealing with values a single register or address can handle.

CHAPTER 12

INTERACTING WITH ASSEMBLY LANGUAGE PROGRAMS

Introduction

Up to this point we've been concentrating on the fundamental instruction set in 651Ø assembly language. Everything we've done has been "locked" into the program. That is, we have no way outside of writing the actual program of affecting what course of action the program will take. When the program is SYSed, it takes a predetermined course the user cannot influence while the program is running. What we've done so far is similar to programming in BASIC without INPUT or GET statements. In this chapter, we're going to turn our attention to some simple I/O (input/output) routines. These will give you some real programming power, and you won't have to learn any new opcodes.

All commercial software "interacts" with the user. When you play an arcade game or use a word processor, your input affects what the program does. For example, in arcade games, the program takes its input from paddles or joysticks. If you press a button, your game fires a missile, and if you move the joystick to the left, your character moves to the left. Similarly, in word processing, your keyboard is the primary input device. Your program stores the in-

formation supplied by the keystrokes, and that information will be different depending on what keys you hit.

Fortunately, there are several built-in routines for handling I/O, and while these routines may use some very complicated and sophisticated code, all you are going to have to do is to know when to JSR to these subroutines. There are a number of "formulas" you'll have to learn since jumping to these subroutines can affect other parts of your program, but these formulas are fairly simple and direct. Moreover, we will concentrate only on those subroutines that affect input from the keyboard, joysticks or game paddles. We will not deal with tape or disk I/O. Since we have already dealt extensively with output in our examples of printing characters to the screen, most of this chapter will concentrate on input.

READING INPUT FROM THE KEYBOARD

Have you ever wondered what actually happens when you press a key? You know that your computer handles everything in binary configurations, and so if you press the 'K' key, a big 'K' doesn't go floating into your computer and up to the screen. We'll break down what happens when you press the letter 'K' to show you the path from the keyboard to your screen.

1. Scan the keyboard.
2. Get the character from the keyboard and put it in the A register
3. See if the key is null (no key has been pressed)
4. If the key is null, go back to Step 1 and scan again.
5. Print the character in the A register to the screen.

Since we have sent ASCII characters from the A register to the screen by storing them on the screen or JSRing to an output routine, we already know how to do the last part. What we need is a routine to scan the keyboard and to get it from the keyboard to the accumulator.

To do that we will use Kernal routines, SCNKEY and GETIN. The SCNKEY subroutine is located at $FF9F (65439) and the GETIN routine at $FFE4 (655Ø8). For output, instead of using

$E716, we'll use CHROUT (Character Out) at $FFD2 (65490). Thus, our sequence of subroutines will be:

1. SCNKEY - Scan the keyboard
2. GETIN - Put the key value in A register
3. CHROUT - Output the character to the screen

Knowing that, we can try out a routine that will print characters to the screen based on keyboard input:

= = LABEL YOUR SUBROUTINES = =

With most assemblers, (not the Kids' Assembler, though) it is possible to define and label subroutines. Then when you want to use a subroutine, instead of doing a JSR to the specific address, the JSR is to the subroutine label. For example, instead of,

JSR $E544

you can,

JSR CLEAR

This method is more descriptive and easier to remember than memorizing all the different addresses that contain the subroutines. Unfortunately the Kids' Assembler does not have a label field, but most other assemblers do.

GENERAL - KEYBOARD TO SCREEN I/O

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore     * = $C000}

{Commodore}
CLEAR          = $E544
SCNKEY         = $FF9F
```

```
GETIN           =$FFE4
CHROUT          =$FFD2

{Merlin}
CLEAR           EQU     $E544
SCNKEY          EQU     $FF9F
GETIN           EQU     $FFE4
CHROUT          EQU     $FFD2

                JSR     CLEAR
SCAN            JSR     SCNKEY  ;Look to see if
                                key is pressed
                JSR     GETIN   ;Put key value in
                                accumulator
                BEQ     SCAN    ;Compare with
                                zero
                JSR     CHROUT  ;If not zero print
                                to screen
                RTS
{Commodore      .END}
```

Scanning Keyboard

KIDS' ASSEMBLER - KEYBOARD TO SCREEN I/O

ADRS	OPCODE	OPERAND
49152	JSR	$E544
49155	JSR	$FF9F
49158	JSR	$FFE4
49161	BEQ	49155
49163	JSR	$FFD2
49166	RTS	

The program involves only two instructions, not counting the RTS. Comparing the code in the Kids' Assembler with the assemblers using labels, you can see how much clearer it is using labels. When we start doing more forward jumps and branches, the labels are almost indispensable. (Be sure to note the different ways that the Commodore assembler and Merlin handle labels.)

Since the program only printed out a single character before ending, we didn't get a chance to do much. What we need is a program to keep reading the keyboard until a certain key is pressed to let us know it's time to quit. We'll write a program that prints the key pressed to the screen until RETURN is pressed. This will involve a double comparison.

1. Compare key with null
2. Compare key with RETURN.

When you run this program, be sure to try out your CTRL keys was well as you regular keys. Turn on the RVS (reverse) and take it through its paces. (If you're smart and you're using a decent editor/assembler, instead of rewriting the whole thing from scratch, you'll just edit the last program to make this one. From this point on, especially when you're writing your own original programs, you will begin to see the severe disadvantages of the Kids' Assembler. Instead of blowing your money on an arcade game, next time you got a few bucks to spend, get a good assembler.)

GENERAL - READ RETURN

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}

{Commodore}
CLEAR           = $E544
SCNKEY          = $FF9F
GETIN           = $FFE4
CHROUT          = $FFD2
```

```
{Merlin}
CLEAR       EQU     $E544
SCNKEY      EQU     $FF9F
GETIN       EQU     $FFE4
CHROUT      EQU     $FFD2

            JSR     CLEAR
SCAN        JSR     SCNKEY
            JSR     GETIN
            BEQ     SCAN
            CMP     #$ØD        ;Compare with
                                ASCII for
                                RETURN (13)
            BEQ     END         ;Jump to end of
                                program if
                                RETURN is
                                pressed
            JSR     CHROUT
            JMP     SCAN
END         RTS
{Commodore  .END}
```

KIDS' ASSEMBLER -

READ RETURN

ADRS	OPCODE	OPERAND
49152	JSR	$E544
49155	JSR	$FF9F
49158	JSR	$FFE4
49161	BEQ	49155
49163	CMP#	$ØD
49165	BEQ	49173
49167	JSR	$FFD2
4917Ø	JMP	49155
49173	RTS	

Now we can see how we can have one of two branches using the keyboard. If we press any key other than RETURN, we'll get the character or the special effects, such as color change or reverse, on the screen. We loop through the SCAN very much like we'd use the GET statement in BASIC.

```
10 GET A$ : IF A$ = "" THEN 10
```

We just keep looping until we get something other than a null. Then, like BASIC, we use the IF/THEN logic to either print the character and go get another one or quit. The same thing in BASIC would look like the following:

```
10 GET A$ : IF A$ = "" THEN 10
20 IF A$ = CHR$(13) THEN END
30 PRINT A$ : GOTO 10
```

Now that we have learned how to use IF/THEN logic with information from the keyboard and can branch in more than one direction, let's use several branches. While we're at it, we'll also supply a prompt and do something other than just printing characters to the screen. A simple routine we already have used is changing the background color on the screen with a JSR to $D021. Instead of entering the color value, we'll enter a letter value for the color and have a subroutine supply the color value. (We're really getting hot.)

First of all, how do we make a prompt? Since the prompts tell us what to do, they're very important in programs. In BASIC, using the GET statement, we simply use PRINT. For example,

```
10 PRINT "PRESS A KEY "
20 GET A$ : IF A$ = "" THEN 20
30 PRINT A$ : GOTO 10
```

asks you to PRESS A KEY and then prints it to the screen.

Since we know that whatever is in the accumulator will be printed to the screen with a JSR to CHROUT ($FFD2) all we have to do is to load up the accumulator with our prompt and print it to the

screen. We're going to change the color of the screen; so we'll use COLOR? as our prompt.

Once we have the prompt, we can use our SCNKEY and GETIN subroutines to read the keyboard and put the results in the accumulator. Now, since we want to change the background color, we will JSR to $DØ21. However, the background colors Ø-15 require CTRL keys, and we want to use regular keys. To keep it relatively simple, we'll just change to background colors to (R)ed or (G)reen. If 'R' is pressed, the background will turn red and 'G' will turn it green. Also, to get out of the program, RETURN will cause an exit. Otherwise, the program will do nothing. Thus, we will have branches on the following:

1. IF RETURN (ASCII = $ØD or 13) is pressed THEN goto the end of the program.

2. IF an R (ASCII = 82) is pressed THEN put a 2 in location $DØ21.

3. IF an G (ASCII = 71) is pressed THEN put a 5 in location $DØ21.

Using the labels END, RED and GREEN it is very simple to indicate where our branches are going and what they are doing. This is why the Kids' Assembler is difficult. You have to figure out the address to branch *ahead* before that address is on the screen. This means that you have to determine the number of bytes that will be used in the program lines so that you will have the right address on a forward branch. Using the LABEL field, however, all you have to do is to put in the label name in the operand, and when you get to the subroutine, use that label at the beginning of the line. (Using the Kids' Assembler, though, you'll really understand what's going on inside your machine. This will turn you into a "mean" assembly language programmer and make programming with a good assembler a snap.)

GENERAL - RED/GREEN BACKGROUND

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------------------------------------------

{Merlin         ORG     $C000}
{Commodore      * = $C000}

; ***********************************************
; *                                             *
; *          RED/GREEN BACKGROUND               *
; *                                             *
; ***********************************************

{Commodore}
CLEAR           = $E544
SCNKEY          = $FF9F
GETIN           = $FFE4
CHROUT          = $FFD2
BKGND           = $D021

{Merlin}
CLEAR           EQU     $E544
SCNKEY          EQU     $FF9F
GETIN           EQU     $FFE4
CHROUT          EQU     $FFD2
BKGND           EQU     $D021
                JSR     CLEAR
                LDA     #67         ;C
                JSR     CHROUT
                LDA     #79         ;O
                JSR     CHROUT
                LDA     #76         ;L
                JSR     CHROUT
                LDA     #79         ;O
                JSR     CHROUT
                LDA     #82         ;R
                JSR     CHROUT
                LDA     #63         ;?
                JSR     CHROUT
                LDA     #0          ;Null the
                                    accumulator
```

```
SCAN        JSR
            SCNKEY
            JSR     GETIN
            BEQ     SCAN
            CMP     #$0D      ;Was RETURN
                              pressed
            BEQ     END       ;If so, goto END
            CMP     #82       ;Was R
                              pressed?
            BEQ     RED       ;If so, goto RED
            CMP     #71       ;Was G pressed
            BEQ     GREEN     ;If so, goto
                              GREEN
            JMP     SCAN      ;If none of the
                              above, go get
                              another key
RED         LDA     #2        ;Color code for
                              RED
                              background
            STA     BKGND
            JMP     SCAN
GREEN       LDA     #5        ;Color code for
                              GREEN
                              background
            STA     BKGND
            JMP     SCAN      ;Go get another
                              key
END         RTS
{Commodore  .END}
```

KIDS' ASSEMBLER - RED/GREEN BACKGROUND

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     JSR     $E544
49155     LDA#    67
49157     JSR     $FFD2
49160     LDA#    79
49162     JSR     $FFD2
49165     LDA#    76
```

```
49167   JSR    $FFD2
49170   LDA#   79
49172   JSR    $FFD2
49175   LDA#   82
49177   JSR    $FFD2
49180   LDA#   63
49182   JSR    $FFD2
49185   LDA#   0
49187   JSR    $FF9F
49190   JSR    $FFE4
49193   BEQ    49187
49195   CMP#   $0D
49197   BEQ    49226
49199   CMP#   82
49201   BEQ    49210
49203   CMP#   71
49205   BEQ    49218
49207   JMP    49187
49210   LDA#   2
49212   STA    $D021
49215   JMP    49187
49218   LDA#   5
49220   STA    $D021
49223   JMP    49187
49226   RTS
```

NOTICE: If you do not have a joystick, you can skip this next section. However, you may want to go over the part on using the EOR instruction.

JOYSTICK CONTROL

Using GETIN we can take the value of a keypress and put it in the accumulator. With the joystick however, we do not have a SCNKEY and GETIN routine. Therefore, we will have to build our own. We'll look at reading the joystick in Port 1 and fire button only. However, Port 2 is read in the same way as Port 1 except from a different address. Therefore, if you know how to read the joystick in Port 1, reading it Port 2 is essentially the same. For Port 1, we

read the value at address $DCØ1 (56321) and for Port 2, $DCØØ(5632Ø). Let's now look to see what happens when you use the joystick and the register at $DCØ1.

Register at $DCØ1

7	6	5	4	3	2	1	Ø	◄-Bit number
X	X	X	F	R	L	D	U	◄-Switch read

X = Unused by joystick
F = Fire button
R = Joystick right
L = Joystick left
D = Joystick down
U = Joystick up
Not pressed = 1
Pressed = Ø

For example, if the joystick is in a neutral position, all switches are read as "not pressed." Therefore, the register would look like the following:

Register at $DCØ1

7	6	5	4	3	2	1	Ø	◄-Bit number
X	X	X	F	R	L	D	U	◄-Switch read
1	1	1	1	1	1	1	1	◄-Bit configuration for neutral

That's simple enough to read since the register is filled up. We know a single 8 bit register can only hold 255 ($FF) so when the joystick is in neutral, the value read at $DØC1 is 255 or $FF. Now, let's take a look at what happens when we move the joystick to the left.

Register at $DCØ1

7	6	5	4	3	2	1	Ø	◄-Bit number
X	X	X	F	R	L	D	U	◄-Switch read
1	1	1	1	1	Ø	1	1	◄-Bit configuration for left.

When we press the joystick to the left, Bit #2 is turned off to give us the binary value, 111111Ø11. Unless you're a lot better at reading binary numbers than I am, that's not easy to figure out. The value is 251 ($FB), but I had to do a lot of computations to get it. There's got to be a simpler way to read the $DCØ1 register.

The EOR Instruction

To make things easier for ourselves, we're going to learn a new instruction, EOR. This instruction is the mnemonic for "Exclusive OR." This instruction compares bits to see the differences. If there is a difference between two bits, you get a "1", and if there are no differences, you get "Ø." This serves to "mask" or "filter" values so that they can be handled easier. To see how it works, let's take our last example and see what it looks like EORed with $FF.

Register at $DCØ1

7	6	5	4	3	2	1	Ø	◄-Bit number
X	X	X	F	R	L	D	U	◄-Switch read
1	1	1	1	1	Ø	1	1	◄-Bit configuration for left.
1	1	1	1	1	1	1	1	◄-$FF bit configuration
Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	◄-EORed with $FF

Now that's a lot easier to figure out. Instead of 251, we have a value of 4. Since bits 7,6 and 5 are not used, we can do relatively quick translations of binary to decimal or hex:

```
16  8  4  2  1  ◄-Multiple if "on."
-----------------------------------------------------------------------------
 4  3  2  1  0  ◄-Bit number
 F  R  L  D  U
 1  0  1  1  1
 1  1  1  1  1  ◄-$FF
-----------------------------------------------------------------------------
 0  1  0  0  0  ◄-EORed value
```

Looking at the above example, what is the value when the joystick is to the right? If you think the value after EOR with $FF is 8 then you are right. If the fire button is pushed, what would the value be? It would be 16. With the joystick in neutral, we saw that the value was $FF or 255 - all eight bits have a "1". After EOR with $FF you would have the following:

Register at $DC01

```
 7  6  5  4  3  2  1  0  ◄-Bit number
 X  X  X  F  R  L  D  U  ◄-Switch read
-----------------------------------------------------------------------------
 1  1  1  1  1  1  1  1  ◄-Bit configuration
                          for left.
 1  1  1  1  1  1  1  1  ◄-$FF bit
                          configuration
-----------------------------------------------------------------------------
 0  0  0  0  0  0  0  0  ◄-EORed with $FF
```

Now, neutral is "0", which is a lot easier to remember.

What happens when your joystick is at an angle, such as up-left? Well, let's just stick the numbers in and see what we get.

```
16  8  4  2  1  ◄-Multiple if "on."
-----------------------------------------------------------------------------
 4  3  2  1  0  ◄-Bit number
 F  R  L  D  U
-----------------------------------------------------------------------------
 1  1  0  1  0
 1  1  1  1  1  ◄-$FF
-----------------------------------------------------------------------------
 0  0  1  0  1  ◄-EORed value
```

A simple calculation shows that the up-left EORed $FF value to be 5. Thus, we can determine the "angle" values of the joystick in addition to the up, down, left and right directions.

You don't have to figure out these values every time you sit down to program, just use the following values as a quick look-up of the EORed joystick values.

Joystick Values EORed with $FF

0 - Neutral
1 - Up
2 - Down
4 - Left
5 - Up Left
6 - Down Left
8 - Right
9 - Up right
10 - Down right
16 - Fire button pressed

The following program will let you see the joystick values on your screen. Remembering that the neutral position is zero, we'll have to add 48 to that value to print the ASCII character "0" to the screen. (ASCII 48 = "0" character.) Thus, as you move your joystick around you will see the actual EORed number in the $DC01 register. By using LDA $DC01 and then EORing the accumulator value with $FF, we put the joystick value into the accumulator. Then by using ADC #48 to add the decimal value 48 to the accumulator and CHROUT, we print it to the screen.

GENERAL - JOYSTICK VALUES

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}
```

```
;*********************************************
;*                                           *
;*            JOYSTICK VALUES                *
;*                                           *
;*********************************************

{Commodore}
JSTICK      =$DC01
OFSET       =$C100
CLEAR       =$E544
FIRE        =$C102
CHROUT      =$FFD2

{Merlin}
JSTICK      EQU   $DC01
OFSET       EQU   $C100
CLEAR       EQU   $E544
FIRE        EQU   $C102
CHROUT      EQU   $FFD2
            JSR   CLEAR
            LDA   #$FF      ;Value to EOR
            STA   OFSET     ;offset
            LDA   #64       ;EORed value of
                            fire button +
                            48
            STA   FIRE      ;Store here for
                            easy reference
START       LDA   JSTICK    ;Read joystick
            EOR   OFSET     ;EOR with $FF
            CLC
            ADC   #48       ;Add 48
            JSR   CHROUT    ;Print modified
                            joystick value to
                            screen
            CMP   FIRE      ;Is it the fire
                            button?
            BEQ   END       ;If it is, then
                            quit
```

```
                JMP     START   ;Go back and
                                read the
                                joystick.
END             RTS
{Commodore      .END}
```

KIDS' ASSEMBLER - JOYSTICK VALUES

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDA#    $FF
49157   STA     $C100
49160   LDA#    64
49162   STA     $C102
49165   LDA     $DC01
49168   EOR     $C100
49171   CLC
49172   ADC#    48
49174   JSR     $FFD2
49177   CMP     $C102
49180   BEQ     49185
49182   JMP     49165
49185   RTS
```

We designed the program so that it would keep printing values to the screen, and your screen fills up with numbers, scrolling different values as you change the direction of the stick. To exit the program, you press the fire button on your joystick.

Now that we can easily read the joystick, let's do something with it. In the next chapter we'll see how to move graphics and sprites with the joystick, but for now we'll just use it to change the background colors. This time, though, we will not use ADC for an offset to the ASCII code. Instead, we'll see what background colors are created with the different joystick positions.

GENERAL - JOYSTICK COLORS

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}


; ********************************************
; *                                          *
; *            JOYSTICK COLORS               *
; *                                          *
; ********************************************


{Commodore}
JSTICK          = $DC01
OFSET           = $C100
CLEAR           = $E544
FIRE            = $C1002

{Merlin}
JSTICK          EQU     $DC01
OFSET           EQU     $C100
CLEAR           EQU     $E544
FIRE            EQU     $C102
                JSR     CLEAR
                LDA     #$FF        ;Value to EOR
                STA     OFSET       ;EOR offset
                LDA     #16         ;EORed value of
                                    fire button
                STA     FIRE        ;Store here for
                                    easy reference

START           LDA     JSTICK      ;Read joystick
                EOR     OFSET       ;EOR with $FF
                CMP     FIRE        ;Fire button
                                    pressed?
```

```
START       LDA     JSTICK    ;Read joystick
            EOR     OFSET     OR with $FF
            CMP     FIRE      ;Fire button
                              pressed?
            BEQ     END       ;If it is then quit
            STA     $D021     ;Put joystick
                              value into
                              background
                              color register
            JMP     START
END         RTS
{Commodore  .END}
```

KIDS' ASSEMBLER - JOYSTICK COLOR

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDA#    $FF
49157   STA     $C100
49160   LDA#    16
49162   STA     $C102
49165   LDA     $DC01
49168   EOR     $C100
49171   CMP     $C102
49174   BEQ     49182
49176   STA     $D021
49179   JMP     49165
49182   RTS
```

Again, we used the fire button to exit the program. Unfortunately, we're left with a black screen. You can change it by using ADC #1 so that instead of the neutral position being black, it will be white.

MAKING MESSAGES : ASC and .BYTE

At the beginning of this chapter we discussed creating prompts. As you may have noticed, it took a lot of code to put a simple prompt like COLOR? on the screen. With most assemblers, there is an

easy way to do it using a pseudo-opcode called ASC, .BYTE or some similar pseudo-opcode. Since the Kids' Assembler does not have such codes, we'll write some simple BASIC programs that will allow you to do make prompts and other messages that can be accessed with assembly language programs.

The ASC and .BYTE instructions operate something like DATA statements in BASIC. The label in the line with the ASC or .BYTE directives serves as the beginning address for the word in the operand field. For example, the following shows the format for the Merlin and Commodore assemblers respectively, with the starting address being MSG:

```
LABEL    OPCODE   OPERAND
-------------------------------------------------------------------------------
MSG      ASC      'MERLIN'
MSG      .BYTE    'COMMODORE'
```

The string in the operand takes up one byte for each character. Therefore, if MSG were address 49170, the string 'MERLIN' would take up 6 addresses (49170-49175) and the string 'COMMODORE', 9 addresses, (49170-49178). Using the X register for an index, simply LDA with MSG indexed by X and print each character to the screen with JSR CHROUT.

First, we'll look at .BYTE and ASC on the Commodore and Merlin assemblers respectively. Both work in the same way, but to avoid confusion, each will be used with a similar but separate example.

Reading ASC & .BYTE

COMMODORE - .BYTE

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
               * = $C000
```

```
;*********************************************
;*                                           *
;*                  .BYTE                    *
;*                                           *
;*********************************************

CLEAR       =$E544
CHROUT      =$FFD2
            JSR     CLEAR
            LDX     #$0
READ        LDA     MSG,X       ;Load one
                                  character
            JSR     CHROUT      ;Print to screen
            CPX     #8          ;See if it is the
                                  length of
                                  message (0-8)
            BEQ     END         ;If it is then end
            INX                 ;Increment X to
                                  read next
                                  character
            JMP     READ
END         RTS
MSG         .BYTE   'COMMODORE'
            .END
```

MERLIN - ASC

```
LABEL       OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
            ORG     $C000

*********************************************
*                                           *
*                  ASC                      *
*                                           *
*********************************************
```

```
CLEAR       EQU     $E544
CHROUT      EQU     $FFD2
            JSR     CLEAR
            LDX     #$0
READ        LDA     MSG,X
            JSR     CHROUT
            CPX     #5
            BEQ     END
            INX
            JMP     READ
END         RTS
MSG         ASC     'MERLIN'
```

Another way to read messages is with a "termination symbol." At the end of your message before the closing single quote mark, place a special symbol that is not likely to be part of the message. The pound sign (#) is a good one to use. Then, instead of having to count the characters, you can simply compare the value in the accumulator with the ASCII value for the termination symbol. For example, instead of having,

```
ASC 'MERLIN'
```

we would put,

```
ASC 'MERLIN#'
```

In that way when ASCII 35, the value for the pound sign (#) is loaded into the accumulator, your program branches out of the READ/PRINT routine. Take a look at the following example on Merlin to see how it works.

MERLIN - ASC RELATIVE

```
LABEL       OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
            ORG     $C000
```

```
**********************************************
*                                            *
*                ASC RELATIVE                *
*                                            *
**********************************************

CLEAR       EQU     $E544
CHROUT      EQU     $FFD2
            JSR     CLEAR
            LDX     #$0
READ        LDA     MSG,X     ;Load one
                              character from
                              MSG
            CMP     #35       ;Is it the pound
                              sign yet?
            BEQ     END       ;If so then end.
            JSR     CHROUT
            INX
            JMP     READ
END         RTS
MSG         ASC     'MERLIN#'
```

Using this method, you can easily create prompts and other messages to be output to the screen. Whenever you need the message, simply read it into the accumulator and throw it out to the screen. Using several descriptive labels, different messages can be accessed and printed depending on where your program branches.

MESSAGE MAKER FOR KIDS' ASSEMBLER

It's a pain in the neck to do a mile of LDA#'s and JSR $FFD2's on the Kids' Assembler to get a message out. Since the Kids' Assembler doesn't have ASC or .BYTE, the following BASIC program gives you a way to stick your message up in memory before you start writing an assembly language program. Essentially, it creates a table with the ASCII values of your message. Be sure to put the message somewhere out of the way of *both* your assembly language program *and* BASIC. If you're working up in the 49152 area, put the message table up around 49200 or 49300 for short programs or in 828 for longer programs if you're not using a

cassette. Note where you put your message, and then when you write an assembly language program, you simply read that area of memory for your message. The BASIC program automatically ends the message with ASCII 35, the value for a pound sign (#); so don't put any other pound signs in your message or prompt. A second BASIC program checks your message for you.

MESSAGE MAKER

```
10 PRINT CHR$(147)
20 Y = 1
30 INPUT"START TABLE ";TS
40 INPUT"MESSAGE ";MS$
50 FOR X = TS TO TS + LEN(MS$)-1
60 S$ = MID$(MS$,Y,1)
70 P = ASC(S$)
80 POKE X,P : Y = Y + 1 : NEXT
90 POKE X,35
```

MESSAGE CHECKER

```
10 INPUT"START ADDR ";SA
20 P = PEEK(SA) : IF PEEK(SA) = 35 THEN END
30 PRINT CHR$(P); : SA = SA + 1 : GOTO 20
```

To test this system, use 49200 as the start address for your message. (Write your name or something original like that.) Once you've checked to see the message is in place with your MESSAGE CHECKER program, enter the following and then execute it:

KIDS' ASSEMBLER - MESSAGE READ/PRINT

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------------------
49152    JSR     $E544
49155    LDX#    0
49157    LDA-X   49200
49160    CMP#    35
49162    BEQ     49171
49164    JSR     $E716
```

```
49167    INX
49168    JMP     49157
49171    RTS
```

Now that you see the principle involved in making and storing ASCII messages to be retrieved from an assembly language program, we'll put together a BASIC program that will make it even easier. Rather than having to write a new message or prompt from our BASIC program every time we need one, why not write and save little "message tables" we can use whenever we need them. In that way, we can load the messages we're going to need into memory, just as we would any assembly language program. By keeping track of different start addresses for each message table and making sure they do not overlap, we can have a whole dictionary of useful messages and prompts. The following program will let you enter a message, and then it will save it. Each message will end with a "termination character", the pound sign (#), so that you can use the CMP# compare-and-branch routine we examined. In that way, you won't have to memorize the length of the message; only its starting address.

MESSAGE MAKER/SAVER

```
10 PRINT CHR$(147)
20 Y = 1
30 INPUT"START TABLE ";TS
40 INPUT"MESSAGE ";MS$
50 FOR X = TS TO TS + LEN(MS$) - 1
60 S$ = MID$(MS$,Y,1)
70 P = ASC(S$)
80 POKE X,P : Y = Y + 1 : NEXT
90 POKE X,35
100 SA = TS : MN$ = MS$
110 MN$ = "0:" + MN$ + STR$(SA) + ",P,W"
120 LB = SA - INT(SA/256)*256
130 HB = INT(SA/256)
140 OPEN2,8,2,MN$
150 PRINT#2,CHR$(LB) + CHR$(HB)
160 P = PEEK(SA)
```

```
170 PRINT#2,CHR$(P)
180 IF P=35 THEN 200
190 SA=SA+1 : GOTO 160
200 CLOSE2
210 END
```

SUMMARY

This chapter has shown you how to write assembly language programs that interact with the user. Using either the keyboard or joystick, the programs respond to the user's actions. Then, in turn, by outputting information to the user in the form of prompts and messages, the user knows what to do next. We spent most of our time with ASCII messages since they most clearly demonstrate what's going on in your program and computer. However, as we saw with changing the background colors, there's more we can do with the keyboard and joystick. In the next chapter, we will see how to manipulate all kinds of graphics with interactive programs.

CHAPTER 13

HOT GRAPHICS

Introduction

This chapter will be a lot of fun. From BASIC, you probably have found that a good deal of work with graphics involves POKEs and PEEKs. In other words, most of the really interesting graphics requires machine language programming from BASIC. As a result of having dealt with machine code and graphics already, you will find what we're doing in this chapter pretty familiar. However, instead of POKEing and PEEKing, we'll use our assembly language instructions to do much the same thing and do it a lot faster. We'll divide the chapter into two major sections:

1. Low resolution color and graphics
2. Animation

In the first section, we'll examine color control and using and changing characters. Since we've already used a lot of examples with background, border and character colors, color control with assembly language will be familiar to you. We will see how by using various characters, we can create low resolution graphics.

The second section will cover animation. In BASIC you may have discovered that movement is often slow and jerky, but in assembly language you will actually have to slow down animated objects so that you can see them! We'll have a separate discussion of sprite animation in the next chapter.

As an extra machine language related matter, we'll also see how to save a screen to disk as a PRG file. This will allow you to LOAD a screen directly from the disk without having to RUN or SYS The program.

LOW RESOLUTION GRAPHICS

Low resolution graphics are created using the characters on your keyboard. The pattern of those characters can be thought of as blocks appearing on your screen. Perhaps the best way to see one of these blocks is to print an inverse space to your screen.

```
POKE 55296,1 : POKE 1024,224
```

That stores the color white in location 55296 and the inverse character for a SPACE in screen location 1024.

By lining up the various blocks, changing their color to what you want, you can create low resolution figures. To get started, let's draw a couple of parallel lines in different colors:

GENERAL

```
LABEL          OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin        ORG     $C000}
{Commodore * = $C000}
```

```
; ********************************************
; *                                          *
; *            LOW-RES LINES                 *
; ********************************************
;
```

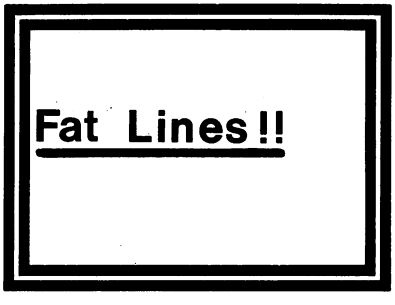
{Commodore}

```
BAR1    =55416
BAR2    =55496
LIN1    =1144
LIN2    =1224
CLEAR   =$E544
```

{Merlin}

```
BAR1    EQU   55416
BAR2    EQU   55496
LIN1    EQU   1144
LIN2    EQU   1224
CLEAR   EQU   $E544
        JSR   CLEAR
        LDX   #$0
        LDY   #2
START   LDA   #224
        STA   LIN1,X   ;1st line of
                       spaces
        STA   LIN2,X   ;2nd line of
                       spaces
        TYA
        STA   BAR1,X   ;1st color of line
        CLC
        ADC   #5       ;Change color
                       with ADC
        STA   BAR2,X   ;2nd color of
                       line
        INX
        CPX   #39
        BNE   START
        RTS
```

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------------------------
49152    JSR     $E544
49155    LDX#    $Ø
49157    LDY#    2
49159    LDA#    224
49161    STA-X   1144
49164    STA-X   1224
49167    TYA
49168    STA-X   55416
49171    CLC
49172    ADC#    5
49174    STA-X   55496
49177    INX
49178    CPX#    39
49180    BNE     49159
49182    RTS
```



We used the X register to index our beginning address to get a straight horizontal line. We started our first line in screen memory 1144 and our second one in 1224. For color, we started the corresponding color addresses at 55416 and 55496. By incrementing our X register from Ø to 39, we were able to draw a line across the screen. Also notice how we used the Y register to get our first color. We never changed the value of Y, and so whenever the program encountered the TYA instruction, it kept putting a 2 (red color code) in the accumulator to be stored on the color screen address. To get the second color (2 + 5 = 7, color code for yellow), we used ADC. Of course, we could have simply put LDA #7, but that wouldn't have been as weird.

To draw different kinds of horizontal lines, try substituting different characters for the space. Instead of 224, try 192, 195 226,239,246,248 and 249 to see what happens.

Since horizontal lines are so simple, vertical lines ought to be a snap. Unfortunately, it doesn't work quite that way. With horizontal lines, we could run one all the way across the screen using the X register as an offset from the beginning address. However, with vertical lines, our address jumps from the top of the screen to the bottom are greater than 255. In fact, there is a difference of 40 between each address we will need. Look down the left side of your screen memory map. It looks like this:

1024
1064
1104
1144
1184
1224
1264
1304
1344
etc.

As you can see, the X register would poop out before it got halfway down the screen. Therefore, we have to use more beginning offset addresses. We'll use five base addresses for both our characters and color. In that way we can stay well within our register limit of 255.

GENERAL - VERTICAL LINES

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}

;*********************************************
;*                                           *
;*                                           *
;*          VERTICAL LINES                   *
;*                                           *
;*                                           *
;*********************************************
```

```
{Commodore}
     BAR1          =55316
     BAR2          =55556
     BAR3          =55796
     BAR4          =56036
     BAR5          =56276
     LIN1          =1044
     LIN2          =1284
     LIN3          =1324
     LIN4          =1564
     LIN5          =1804
     CLEAR         =$E544

{Merlin}
     BAR1          EQU     55316
     BAR2          EQU     55556
     BAR3          EQU     55796
     BAR4          EQU     56036
     BAR5          EQU     56276
     LIN1          EQU     1044
     LIN2          EQU     1284
     LIN3          EQU     1324
     LIN4          EQU     1564
     LIN5          EQU     1804
     CLEAR         EQU     $E544

                   JSR     CLEAR
                   LDX     #$0
     START         LDA     #224
                   STA     LIN1,X
                   STA     LIN2,X
                   STA     LIN3,X
                   STA     LIN4,X
                   STA     LIN5,X
                   LDA     #2
                   STA     BAR1,X
                   STA     BAR2,X

                   STA     BAR3,X
                   STA     BAR4,X
```

```
                STA     BAR5,X
                LDY     #0
INXR            INX
                INY
                CPY     #40
                BNE     INXR
                CPX     #240
                BNE     START
                RTS
```

KIDS' ASSEMBLER - VERTICAL LINES

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDX#    $0
49157   LDA#    224
49159   STA-X   1044
49162   STA-X   1284
49165   STA-X   1324
49168   STA-X   1564
49171   STA-X   1804
49174   LDA#    2
49176   STA-X   55316
49179   STA-X   55556
49182   STA-X   55796
49185   STA-X   56036
49188   STA-X   56276
49191   LDY#    0
49193   INX
49194   INY
49195   CPY#    40
49197   BNE     49193
49199   CPX#    240
49201   BNE     49157
49203   RTS
```

Another way to work with low resolution graphics is with your CHROUT routine. Instead of having to deal with both color and character screens, you can just deal with the characters. The screen

addresses make it handy to place blocks where you want them, but with CHROUT, you can use the PLOT subroutine to place your graphics. To see how this works, let's make a diagonal line.

In using the PLOT subroutine, you have to be sure to LDA your character to be printed on the screen AFTER you PLOT. This is because the accumulator can be scrambled by the PLOT subroutine. Also, be sure to CLC before you JSR to PLOT ($FFFØ), otherwise you may end up reading and not setting the plot location. (To read the plot location, you set the carry flag with SEC.)

GENERAL - DIAGONAL LINES

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------------------------------------------
{Merlin         ORG     $CØØØ}
{Commodore      * = $CØØØ}

; ********************************************
; *                                          *
; *          DIAGONAL PLOT                   *
; *                                          *
; ********************************************

{Commodore}
CHROUT          = $FFD2
CLEAR           = $E544
PLOT            = $FFFØ

{Merlin}
CHROUT          EQU     $FFD2
CLEAR           EQU     $E544
PLOT            EQU     $FFFØ
                JSR     CLEAR
                LDX     #Ø
                LDY     #Ø
                LDA     #18         ;Character for
                                    inverse.
                JSR     CHROUT
START           CLC
```

```
JSR     PLOT
LDA     #32     ;Character for
                space.
JSR     CHROUT
INX             ;Next row
INY
                ;Next column
CPX     #25     ;Is it at the
                bottom row
                yet?
BNE     START   ;If not go back
                and do it again.
RTS
```

KIDS' ASSEMBLER - DIAGONAL PLOT

```
ADRS    OPCODE  OPERAND
-------------------------------------------------------------------------------
49152   JSR     $E544
49155   LDX#    Ø
49157   LDY#    Ø
49159   LDA#    18
49161   JSR     $FFD2
49164   CLC
49165   JSR     $FFFØ
49168   LDA#    32
4917Ø   JSR     $FFD2
49173   INX
49174   INY
49175   CPX#    25
49177   BNE     49164
49179   RTS
```

Admittedly, low resolution graphics aren't much for detailed drawings, but they're a lot of fun. Change the background and border colors by inserting values between Ø and 15 into $DØ21 and $DØ2Ø. (These are the routines we used in some of our previous examples. The hexadecimal numbers are easier to remember than the decimal ones once you get used to them.)

One way to have finer resolution in low resolution graphics is to use characters that have the kinds of lines you want in your graphic display. For example, if you want to have a nice thin diagonal line in the DIAGONAL PLOT program above, just remove the lines that inverse the screen (LDA #18 and JSR CHROUT), and use the value 1Ø9 instead of 32 to output to the screen. ASCII 1Ø9 is a left-leaning angle. Put them together in a diagonal plot, and you will have a very straight line instead of the "staircase" we got.

SAVING PLOTTED GRAPHICS

When you draw low-resolution figures with the PLOT subroutine, you can save them as PRG files and load them directly to your screen. Actually, it takes a lot less disk space to save your graphics as machine language files and then load and SYS them, but there are some applications where you might want to "overlay" one set of graphics with another. Besides, it's interesting and useful to know. (You can save anything on the screen, actually, and so this method can be used to save text as well.)

Essentially, what you do is to scan the entire screen, and then use the screen addresses (1Ø24-2Ø23) as your machine code file. Then when you LOAD "GRAPHICFILE",8,1 your graphic will be on the screen. Because your load address *is* the screen, it appears immediately without the necessity of having to SYS it.

The following BASIC program is to be used in conjunction with a machine language program you have put in memory. For example, to save the DIAGONAL PLOT figure (not program) as a PRG file, you first load your program into memory like you would normally do. However, instead of SYSing the program at this time, enter NEW and RETURN. Then LOAD and RUN the following program. It will first SYS the program in memory so that the only thing on your screen is the graphic figures. Then, it will treat the entire screen as a machine language program and put it on disk just as any other machine language program would be. They take up about four blocks of disk space compared to one block taken up by small assembly language programs such as DIAGONAL PLOT. Here's a summary of the steps to use:

Step 1. Either write a machine language graphic program in your assembler or load one from disk.

Step 2. Enter NEW and RETURN.

Step 3. LOAD the SAVE PLOT program and RUN it.

SAVE PLOT

```
10 INPUT "NAME OF FILE TO SAVE"; NF$
20 NF$ = "0:" + NF$ + ",P,W"
30 INPUT "START ADDRESS";TER
40 SYSTER
50 BA = 1024 : EA = 2023
60 LB = BA-INT(BA/256)*256
70 HB = INT(BA/256)
80 OPEN2,8,2,NF$
90 PRINT#2,CHR$(LB) + CHR$(HB)
100 FOR X = BA TO EA
110 P = PEEK(X)
120 PRINT#2,CHR$(P)
130 NEXT
140 CLOSE2
```

Take a look at Line 40. You probably know that there's no such BASIC command as SYSTER (pronounced 'sister'), but there it is, and everything works fine. It's just a way to do weird things with your computer. Since the starting address of your machine program in memory is INPUT in the variable TER, what it actually does is to SYS the variable TER. Put them together, and there you have it. (If you think that's dumb, use the variable name BOOMBAH and see what happens.)

Before we go on to the next section, you're probably wondering why we didn't use our program to save program files of graphics created with storage to the screen addresses instead of just ones created with the PLOT and CHROUT subroutines. Since most Commodore 64's require that a corresponding color be included when you STA an ASCII value in a screen address, we would have

had to have two files saved; one for the screen and one for the color. You can do it if you want, but it didn't seem to be worth the bother.

ANIMATION

Animation in assembly language requires some programming concentration, but it's so spectacular when you're finished, it's worth it. All animation is based on the illusion of drawing a figure in one place, erasing it, and drawing it in another place. It works just like cartoons in the movies. A figure is drawn, shown for an instant on the screen, and then it is removed (erased) and another figure is put in its place.

Your computer makes animation very simple since it can create and erase figures in different places on the screen at high speeds. In fact, with assembly/machine language, it is often too fast. To see the difference in speed between a BASIC and assembly language program doing the same thing, look at try the following programs.

BASIC ANIMATION

```
10 PRINT CHR$(147)
20 FOR X = 1 TO 10 : PRINT : NEXT
30 FOR X = 1 TO 39 : PRINT CHR$(113); : PRINT
CHR$(157); CHR$(32);
40 NEXT
50 PRINT CHR$(113)
```

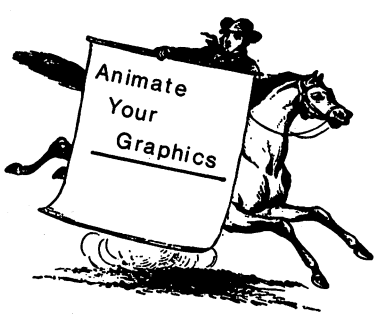
The ball really sails along in BASIC. A little flicker maybe, but it moves nicely. Try it in assembly language now.

*NOTE: We used different beginning addresses for Merlin and the Commodore assemblers. The default ORG for Merlin is $8000, and if we use it, we can test assembled programs in the monitor before we save the program to disk. From the Edit Mode, enter MON {RETURN} and you will be in the monitor, indicated by a '$' prompt. To test the program, just enter 8000g. All values in the Merlin monitor are assumed to be hexadecimal. We have avoided the $8000 address for the beginning of your object code because you might be using a

plug-in ROM. However, as we get into more complex and longer programs, you'd better start using the Merlin monitor for debugging your programs. If you have a ROM installed, take it out before you start your assembly language programming with Merlin. If your program ORG is at $C000, it may interfere with Merlin or the monitor.

GENERAL

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $8000}
{Commodore      * = $C000

;*****************************************
;*                                       *
;*            ANIMATION 1                *
;*                                       *
;*****************************************

{Commodore}
    CLEAR       = $E544
    CHROUT      = $FFD2
    PLOT        = $FFF0
    EX          = $C200
    WHY         = $C202
    BALL        = $C204
    SPACE       = $C206
```




```
{Merlin}
    CLEAR       EQU     $E544
    CHROUT      EQU     $FFD2
    PLOT        EQU     $FFF0
    EX          EQU     $8200
    WHY         EQU     $8202
    BALL        EQU     $8204
    SPACE       EQU     $8206
```

```
        JSR     CLEAR
        LDX     #10         ;Set to row 10.
        STX     EX
        LDY     #0          ;Set to column
                            0.
        STY     WHY
        LDA     #113        ;ASCII value for
                            ball
        STA     BALL
        LDA     #32         ;ASCII value for
                            space
        STA     SPACE

START   LDX     EX
        LDY     WHY
        CLC
        JSR     PLOT        ;Plot the ball
        LDA     BALL        ;Load the ball
        JSR     CHROUT      ;Print the ball
        LDX     EX          ;Load X register
                            with last row
                            plot
        LDY     WHY         ;Load Y register
                            with last
                            column plot
        CLC
        JSR     PLOT        ;Plot the space
        LDA     SPACE       ;Load the space
        JSR     CHROUT      ;Erase the ball
                            with the space
        INC     WHY         ;Increment the
                            column value
                            (Y)
        LDY     WHY         ;Load the Y
                            register with
                            the next
                            column
        CPY     #38         ;Is it near the
                            last column
        BNE     START       ;If not print and
```

```
                                        erase another
                                        ball
        CLC
        JSR     PLOT
        LDA     BALL
        JSR     CHROUT  ;Put a ball on
                        the screen so
                        there's
                        something left
        RTS
```

KIDS' ASSEMBLER - ANIMATION 1

ADRS	OPCODE	OPERAND
49152	JSR	$E544
49155	LDX#	10
49157	STX	$C200
49160	LDY#	0
49162	STY	$C202
49165	LDA#	113
49167	STA	$C204
49170	LDA#	32
49172	STA	$C206
49175	LDX	$C200
49178	LDY	$C202
49181	CLC	
49182	JSR	$FFF0
49185	LDA	$C204
49188	JSR	$FFD2
49191	LDX	$C200
49194	LDY	$C202
49197	CLC	
49198	JSR	$FFF0
49201	LDA	$C206
49204	JSR	$FFD2
49207	INC	$C202
49210	LDY	$C202
49213	CPY#	38
49215	BNE	49175

```
49217    CLC
49218    JSR     $FFF0
49221    LDA     $C204
49224    JSR     $FFD2
49227    RTS
```

You may have thought you did something wrong. If all you saw was the ball on the right side of the screen after you SYSed the program, you keyed it in correctly. Animation is so fast in assembly/machine language that you can't see the movement unless you slow it down. To do that, we'll put in a PAUSE loop. In fact, we'll have to put in a nested PAUSE loop since even a loop of 255 won't slow the movement down enough. All the pause loop does is to run through an "empty loop" to slow down a program. In this case, we ran through the loop 2550 times. Edit ANIMATION 1 so that it includes the PAUSE loop in ANIMATION 2, and run the program again. Now you can see the ball move smoothly across the screen. To increase or decrease the speed of the ball, increase or decrease the CPY #$0A.

GENERAL - ANIMATION 2

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $8000}
{Commodore      * = $C000}

; ************************************************
; *                                              *
; *                 ANIMATION 2                  *
; *                                              *
; ************************************************

{Commodore}
    CLEAR       = $E544
    CHROUT      = $FFD2
    PLOT        = $FFF0
    EX          = $C200
    WHY         = $C202
```

```
BALL        =$C204
SPACE       =$C206

{Merlin}
CLEAR       EQU     $E544
CHROUT      EQU     $FFD2
PLOT        EQU     $FFF0
EX          EQU     $8200
WHY         EQU     $8202
BALL        EQU     $8204
SPACE       EQU     $8206
            JSR     CLEAR
            LDX     #10
            STX     EX
            LDY     #0
            STY     WHY
            LDA     #113
            STA     BALL
            LDA     #32
            STA     SPACE
START       LDX     EX
            LDY     WHY
            CLC
            JSR     PLOT
            LDA     BALL
            JSR     CHROUT
            LDY     #0          ;Begin pause
                                 loop.
PAUSE1      LDX     #0
PAUSE2      INX
            CPX     #$FE
            BNE     PAUSE2
            INY
            CPY     #$0A
            BNE     PAUSE1      ;End pause
                                 loop.
            LDX     EX
            LDY     WHY
            CLC
            JSR     PLOT
```

```
LDA     SPACE
JSR     CHROUT
INC     WHY
LDY     WHY
CPY     #38
BNE     START
CLC
JSR     PLOT
LDA     BALL
JSR     CHROUT
RTS
```

KIDS' ASSEMBLER - ANIMATION 2

ADRS	OPCODE	OPERAND
49152	JSR	$E544
49155	LDX#	10
49157	STX	$C200
49160	LDY#	0
49162	STY	$C202
49165	LDA#	113
49167	STA	$C204
49170	LDA#	32
49172	STA	$C206
49175	LDX	$C200
49178	LDY	$C202
49181	CLC	
49182	JSR	$FFF0
49185	LDA	$C204
49188	JSR	$FFD2
49191	LDY#	0
49193	LDX#	0
49195	INX	
49196	CPX#	$FE
49198	BNE	49195
49200	INY	
49201	CPY#	$0A
49203	BNE	49193
49205	LDX	$C200

```
49208   LDY     $C202
49211   CLC
49212   JSR     $FFF0
49215   LDA     $C206
49218   JSR     $FFD2
49221   INC     $C202
49224   LDY     $C202
49227   CPY#    38
49229   BNE     49175
49231   CLC
49232   JSR     $FFF0
49235   LDA     $C204
49238   JSR     $FFD2
49241   RTS
```

That was a lot of work to get that crummy ball moving across the screen, and if you used the Kids' Assembler, you might be thinking the French Foreign Legion would be involve less pain. You might even be desperate enough to go back to BASIC. (God forbid!) On the other hand, if you used an assembler with a decent editor, all you had to do was to insert the eight lines for the PAUSE loop. If you have a birthday coming up, it's near Christmas or you have a half of ton of aluminum cans to take to the recycling center, think about getting a good assembler.

EXTERNAL CONTROL OF MOVEMENT

Now that we have seen how to move objects, let's see how to control them with an external device. We'll use the joystick for our examples, but you could do the same thing with the keyboard. Just substitute the SCNKEY and GETIN routines for the joystick ones in the following programs.

The nice thing about using the PLOT subroutine in animation is that you can use the X and Y registers to place things on the screen in sequential locations. However, when you start moving all over the screen, PLOT can really scramble things, especially your brain. Therefore, we will begin using the ASCII code for your cursor control.

CURSOR CONTROL CODES

Up	145	$91
Down	17	$11
Left	157	$9D
Right	29	$1D

We'll use the CHROUT subroutine for moving both the cursor and the character. The trick is in remembering that CHROUT moves everything to the RIGHT. Therefore, when we move down, we actually move down and right with CHROUT. Therefore, we will have to make adjustments when we move in any other direction than right. To get started, we'll make a blinking cursor and move it around the screen. Again, note the different addresses used by Merlin and Commodore assemblers for the starting location.

GENERAL - JOYSTICK CURSOR

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $8000}
{Commodore      * = $C000}

; ***********************************************
;*                                              *
; *            JOYSTICK CURSOR                  *
;*                                              *
; ***********************************************

{Commodore}
CLEAR           = $E544
JSTICK          = $DC01
OFSET           = $C200
FIRE            = $C202
INVERSE         = $C204
NORMAL          = $C206
MARK            = $C208
CHROUT          = $FFD2
```

```
{Merlin}
CLEAR     EQU   $E544
JSTICK    EQU   $DC01
          EQU   $8200
          EQU   $8202
INVERSE   EQU   $8204
NORMAL    EQU   $8206
MARK      EQU   $8208
CHROUT    EQU   $FFD2
          JSR   CLEAR
          LDA   #$FF      OR value
          STA   OFSET
          LDA   #16       ;Fire button
          STA   FIRE
          LDA   #32       ;Space for the
                          cursor
          STA   MARK
          LDA   #18       ;Inverse Code
          STA   INVERSE
          LDA   #146      ;Normal code
          STA   NORMAL
START     LDA   JSTICK
          EOR   OFSET
          CMP   #1        ;Joystick up?
          BEQ   UP
          CMP   #2        ;Joystick
                          down?
          BEQ   DOWN
          CMP   #4        ;Joystick left?
          BEQ   LEFT
          CMP   #8        ;Joystick right?
          BEQ   RIGHT
          CMP   FIRE      ;Fire button
                          pressed?
          BEQ   END       ;If so then end.
CURSOR    LDA   MARK      ;Load the
                          space
          JSR   CHROUT    ;Print the space
          LDA   #157      ;Load the left
                          cursor
```

```
                JSR     CHROUT  ;Back up
                LDA     NORMAL
                JSR     CHROUT  ;Set normal
                LDA     MARK
                JSR     CHROUT
                LDA     #157    ;Back up
                JSR     CHROUT
                LDA     INVERSE
                JSR     CHROUT  ;Set inverse
                JMP     START   ;Go do it again
UP              LDA     #145
                JMP     PRINT   ;Print up cursor
DOWN            LDA     #17
                JMP     PRINT   ;Print down
                                cursor
LEFT            LDA     #157
                JMP     PRINT   ;Print left cursor
RIGHT           LDA     #29
PRINT           JSR     CHROUT
                JMP     CURSOR
END             RTS
{Commodore      .END}
```

KIDS' ASSEMBLER - JOYSTICK CURSOR

ADRS	OPCODE	OPERAND
49152	JSR	$E544
49155	LDA#	$FF
49157	STA	$C200
49160	LDA#	16
49162	STA	$C202
49165	LDA#	32
49167	STA	$C208
49170	LDA#	18
49172	STA	$C204
49175	LDA#	146
49177	STA	$C206
49180	LDA	$DC01
49183	EOR	$C200

```
49186   CMP#    1
49188   BEQ     49244
49190   CMP#    2
49192   BEQ     49249
49194   CMP#    4
49196   BEQ     49254
49198   CMP#    8
49200   BEQ     49259
49202   CMP     $C202
49205   BEQ     49267
49207   LDA     $C208
49210   JSR     $FFD2
49213   LDA#    157
49215   JSR     $FFD2
49218   LDA     $C206
49221   JSR     $FFD2
49224   LDA     $C208
49227   JSR     $FFD2
49230   LDA#    157
49232   JSR     $FFD2
49235   LDA     $C204
49238   JSR     $FFD2
49241   JMP     49180
49244   LDA#    145
49246   JMP     49261
49249   LDA#    17
49251   JMP     49261
49254   LDA#    157
49256   JMP     49261
49259   LDA#    29
49261   JSR     $FFD2
49264   JMP     49207
49267   RTS
```

When you assemble and run this program, you will see a flashing cursor made by the SPACE character being turned normal and inverse rapidly. When you move your joystick, the cursor will be wholly out of control, jumping clear across the screen with a quick movement of your joystick. Just as we saw with animation,

machine language is just too blasted fast. We'll have to slow it down with a delay loop to get single space control over it.

Now that we have seen some general movement principles with our cursor, let's draw with the joystick. This is not "animated" movement in that what we draw is not erased and then re-drawn. However, very similar principles apply when controlling what happens with graphics on the screen. We'll also add a couple of new tricks.

First, we saw in the last program that we had to keep drawing and backing up. This required several LDA's and JSR's throughout the program. Since we know we're going to have to back up, why not make a subroutine that will do that. The subroutine will be accessed just like the built-in subroutines using JSR. However, since we're writing the subroutine as part of our own program, we have to include an RTS to get back to our jump-off point. It works just like GOSUB and RETURN in BASIC.

Second, we're going to put a pause loop in our "joystick scan" to slow things down a bit. In the last program, when you moved the joystick, the cursor hopped all the way across the screen. This was because the scan was so quick that it was able to read the direction of the joystick and jump to the subroutines several times before you could put it in neutral. (In fact, you may want to increase the pause loop we put in this program!)

The program draws low resolution lines on the screen. To keep it simple and a little more flexible, we'll use the joystick LEFT to serve as an eraser. Therefore, if you move the joystick UP, DOWN or RIGHT, a line will be drawn. Move it left, and anything the cursor hits will be erased.

GENERAL

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $8000}
{Commodore     * = $C000}
```

```
;*******************************************
;*                                         *
;*            JOYSTICK DRAW                *
;*                                         *
;*******************************************

{Commodore}
CLEAR       =$E544
JSTICK      =$DC01
OFSET       =$C200
FIRE        =$C202
INVERSE     =$C204
NORMAL      =$C206
MARK        =$C208
CHROUT      =$FFD2
{Merlin}
CLEAR       EQU     $E544
JSTICK      EQU     $DC01
OFSET       EQU     $8200
FIRE        EQU     $8202
INVERSE     EQU     $8204
NORMAL      EQU     $8206
MARK        EQU     $8208
CHROUT      EQU     $FFD2
            JSR     CLEAR
            LDA     #$FF
            STA     OFSET
            LDA     #16
            STA     FIRE
            LDA     #32
            STA     MARK
            LDA     #18
            STA     INVERSE
            LDA     #146
            STA     NORMAL
START       LDA     JSTICK
            EOR     OFSET
            CMP     #1
            BEQ     UP
```

```
                LDA     INVERSE
                JSR     CHROUT
                LDX     #0          ;Begin pause
                                    loop
PAUSE           INX
                CPX     #254
                BNE     PAUSE       ;End pause loop
                JMP     START
UP              JSR     BACK
                LDA     #145
                JMP     PRINT
DOWN            JSR     BACK
                LDA     #17
                JMP     PRINT
LEFT            LDA     #157
                JSR     CHROUT
                JMP     PRINT
PRINT           JSR     CHROUT
                LDA     MARK
                JSR     CHROUT
                JMP     CURSOR
BACK            LDA     #157        ;Subroutine
                JSR     CHROUT
                RTS
END             RTS
{Commodore      .END}
```

KIDS' ASSEMBLER - JOYSTICK DRAW

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     JSR     $E544
49155     LDA#    $FF
49157     STA     $C200
49160     LDA#    16
49162     STA     $C0202
49165     LDA#    32
49167     STA     $C208
49170     LDA#    18
49172     STA     $C204
```

```
49175   LDA#    146
49177   STA     $C206
49180   LDA     $DC01
49183   EOR     $C200
49186   CMP#    1
49188   BEQ     49247
49190   CMP#    2
49192   BEQ     49255
49194   CMP#    4
49196   BEQ     49263
49198   CMP#    8
49200   BEQ     49271
49202   CMP     $C0202
49205   BEQ     49289
49207   LDA     $C208
49210   JSR     $FFD2
49213   JSR     49285
49216   LDA     $C206
49219   JSR     $FFD2
49222   LDA     $C208
49225   JSR     $FFD2
49228   JSR     49285
49231   LDA     $C204
49234   JSR     $FFD2
49237   LDX#    0
49239   INX
49240   CPX#    254
49242   BNE     49239
49244   JMP     49180
49247   JSR     49285
49250   LDA#    145
49252   JMP     49271
49255   JSR     49285
49258   LDA#    17
49260   JMP     49271
49263   LDA#    157
49265   JSR     $FFD2
49268   JMP     49271
49271   JSR     $FFD2
49274   LDA     $C208
```

```
49277    JSR     $FFD2
49280    JMP     49207
49283    LDA#    157
49285    JSR     $FFD2
49288    RTS
49289    RTS
```

That was a long one! Be sure to note the *two* RTS instructions at the end of the program. The first one is to return to the program position that initiated the JSR. It works like RETURN. The second one is to return to BASIC and end the machine level program. Of course the RTS that returns to BASIC does not have to be at the end of the program, but it must be the last RTS encountered in the program flow.

SUMMARY

In this chapter we saw how to work with low resolution graphics and animation. However, we learned a few new tricks in assembly language programming, as well. We wrote our own subroutine that we JSRed and RTSed from and saw how to make delay loops. By combining previous skills, we even made a drawing program with the joystick.

We did some work with color, but not a great deal. This was because we already know how to change the colors to anything we want from examples in previous chapters. Also, since we were doing a good deal of work with new concepts, more color may have been confusing. Besides, I wanted you to test some of your own skills in this area. Experiment with different border, background and character colors with the programs. You'll be surprised by the difference it makes.

In the next chapter, we will examine sprite graphics and sound. There, we will combine new tricks with some of the those we learned in this chapter. We will be learning the fundamentals of arcade game assembly language programming.

CHAPTER 14

BLAZING SPRITES AND MONSTROUS SOUNDS

SPRITE GRAPHICS

If you've worked with sprites in BASIC, you will find working with them in assembly language is easier! Since most of the set-up you had in BASIC involved POKEs and PEEKs, you were actually working with machine language. We all know that assembly language is simpler than machine language; so this ought to be a snap for you BASIC sprite programmers. For those of you who have not worked with sprites, we'll take it a step at a time.

In case you don't know what a sprite is on the Commodore 64, let me explain. Basically, it is a 63 byte graphic image. Each byte is given a value to turn on a configuration of pixels; little dots on the screen. From our discussion of binary math and how the various registers look, let's take a look at a single byte.

7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	1
On	Off	Off	On	On	On	On	On

On your screen, the above configuration would look something like the following if it were magnified:

```
. .....
```

Each sprite is composed of three columns and 21 rows. Each row is made up of three bytes. Since each byte has 8 bits that can be turned on or off, it is clearer to think of a sprite graphic as a 21 by 24 matrix of bits or pixels.

Sprite Matrix

Column 1	Column 2	Column 3	Row
XXXXXXXX	XXXXXXXX	XXXXXXXX	1
XXXXXXXX	XXXXXXXX	XXXXXXXX	2
XXXXXXXX	XXXXXXXX	XXXXXXXX	3
XXXXXXXX	XXXXXXXX	XXXXXXXX	4
XXXXXXXX	XXXXXXXX	XXXXXXXX	5
XXXXXXXX	XXXXXXXX	XXXXXXXX	6
XXXXXXXX	XXXXXXXX	XXXXXXXX	7
XXXXXXXX	XXXXXXXX	XXXXXXXX	8
XXXXXXXX	XXXXXXXX	XXXXXXXX	9
XXXXXXXX	XXXXXXXX	XXXXXXXX	10
XXXXXXXX	XXXXXXXX	XXXXXXXX	11
XXXXXXXX	XXXXXXXX	XXXXXXXX	12
XXXXXXXX	XXXXXXXX	XXXXXXXX	13
XXXXXXXX	XXXXXXXX	XXXXXXXX	14
XXXXXXXX	XXXXXXXX	XXXXXXXX	15
XXXXXXXX	XXXXXXXX	XXXXXXXX	16
XXXXXXXX	XXXXXXXX	XXXXXXXX	17
XXXXXXXX	XXXXXXXX	XXXXXXXX	18
XXXXXXXX	XXXXXXXX	XXXXXXXX	19
XXXXXXXX	XXXXXXXX	XXXXXXXX	20
XXXXXXXX	XXXXXXXX	XXXXXXXX	21

In the above matrix, suppose that the 'x' marks represent a Ø. If we turned on a pixel, it would be a '+'. To draw a character, we just have to put little '+' marks in the shape we want. To keep it simple, we'll draw a cross:

Column 1	Column 2	Column 3	Row
XXXXXXXX	++++++++	XXXXXXXX	1
XXXXXXXX	++++++++	XXXXXXXX	2
XXXXXXXX	++++++++	XXXXXXXX	3
XXXXXXXX	++++++++	XXXXXXXX	4
XXXXXXXX	++++++++	XXXXXXXX	5
++++++++	++++++++	++++++++	6
++++++++	++++++++	++++++++	7
++++++++	++++++++	++++++++	8
++++++++	++++++++	++++++++	9
++++++++	++++++++	++++++++	10
XXXXXXXX	++++++++	XXXXXXXX	11
XXXXXXXX	++++++++	XXXXXXXX	12
XXXXXXXX	++++++++	XXXXXXXX	13
XXXXXXXX	++++++++	XXXXXXXX	14
XXXXXXXX	++++++++	XXXXXXXX	15
XXXXXXXX	++++++++	XXXXXXXX	16
XXXXXXXX	++++++++	XXXXXXXX	17
XXXXXXXX	++++++++	XXXXXXXX	18
XXXXXXXX	++++++++	XXXXXXXX	19
XXXXXXXX	++++++++	XXXXXXXX	20
XXXXXXXX	++++++++	XXXXXXXX	21

Next, to envision our cross as a set of 1's and zero's, let's change the X's to Ø 's and the + 's to 1's.

Column 1	Column 2	Column 3	Row
ØØØØØØØØ	11111111	ØØØØØØØØ	1
ØØØØØØØØ	11111111	ØØØØØØØØ	2
ØØØØØØØØ	11111111	ØØØØØØØØ	3
ØØØØØØØØ	11111111	ØØØØØØØØ	4
ØØØØØØØØ	11111111	ØØØØØØØØ	5
11111111	11111111	11111111	6
11111111	11111111	11111111	7
11111111	11111111	11111111	8
11111111	11111111	11111111	9
11111111	11111111	11111111	10

```
00000000 11111111 00000000    11
00000000 11111111 00000000    12
00000000 11111111 00000000    13
00000000 11111111 00000000    14
00000000 11111111 00000000    15
00000000 11111111 00000000    16
00000000 11111111 00000000    17
00000000 11111111 00000000    18
00000000 11111111 00000000    19
00000000 11111111 00000000    20
00000000 11111111 00000000    21
```

To get our cross into the computer, we will have to change the binary values into decimal or hex so that we can put them into memory. (We could do it with binary numbers, but that would be impossible with the Kids' Assembler.) We know a byte with 00000000 is equal to 0 and a byte with 11111111 is 255 or $FF. To arrange our cross in the correct sequential order, we move from left to right and top to bottom. In the column to the right of our figure, we'll place the decimal values.

```
Column 1 Column 2 Column 3   Value
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
11111111 11111111 11111111 255,255,255
11111111 11111111 11111111 255,255,255
11111111 11111111 11111111 255,255,255
11111111 11111111 11111111 255,255,255
11111111 11111111 11111111 255,255,255
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
```

```
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
00000000 11111111 00000000   0,255,0
```

To place a sprite into memory, you begin with Row 1 Column 1 and move to Row 1 Column 2, Row 1 Column 3, Row 2 Column 1, Row 2 Column 2 etc. and place the values in sequential memory locations. (You snake along in other words.) Thus, in the first two rows, we would place the values into memory in the following order:

Location	Value	Column	Row
#1	0	1	1
#2	255	2	1
#3	0	3	1
#4	0	1	2
#5	255	2	2
#6	0	3	2

Of course, you're not limited to values of 0 or 255. Depending on the bit configuration, your sprite bytes can be any combination of values between 0 and 255. For example, you might have a configuration that looks like the following:

```
00001111 00111100 1111000  15,60,240
00000111 00011000 1110000   7,24,224
00000011 01100110 1100000  3,102,192
```

Use the BINARY - DECIMAL conversion program in Chapter 5 to easily make the conversion from binary to decimal.

Further on in this chapter we will write a "Sprite Assembler" in BASIC that will store your sprites and save them. Before that, though, we must first go over the procedure for getting sprites to work.

SPRITE CREATION

The only way to work with sprites and assembly language is to GET ORGANIZED! Programming sprites is actually quite simple

once you have everything set up in a simple sequence and know what registers to use. To begin, let's outline the basic sequence of programming sprites

SIMPLE SPRITE SEQUENCE

1. Store block number into POINTER.
2. ENABLE Sprite
3. Store sprite COLOR in sprite color register
4. BUILD sprite and store it.
5. Set horizontal HIGH bit to 1 or Ø.
6. MOVE sprite

The above sequence provides the order of events in your program. Assembly language uses the same sequence as BASIC; so there's nothing new about it. Now, let's look at each step closely.

1. Store Block Number into POINTER.

We will be storing our sprites in available memory blocks of 63 bytes each. Therefore, we must find some 63 byte blocks we can use.

Block Number	Addresses:	
11	7Ø4-767	$2CØ-$2FF
13	832-895	$34Ø-$37F
14	869-959	$38Ø-$3BF
15	96Ø-1Ø23	$3CØ-$3FF

The block number refers to the number of 64 byte blocks (not 63 since 64 is evenly divided into 256) below the starting address. For example, Block Ø is at addresses Ø-63, Block 1 at 64-127 and so forth. Without rearranging memory, those four blocks are about as much as we can handle. Therefore, we can build four sprites. (With memory management, you can get up to eight sprites going.)

Now that we know what a block number is, we must store that value in a register that points to the block in which we will store our

sprite. The pointer address begins at 2040 ($7F8). To get the correct pointer for our sprite, we add the sprite number to the base address.

Sprite number	POINTER	Register
Sprite 0	2040	$FF8
Sprite 1	2041	$FF9
Sprite 2	2042	$FFA
Sprite 3	2043	$FFB
Sprite 4	2044	$FFC
Sprite 5	2045	$FFD
Sprite 6	2046	$FFE
Sprite 7	2047	$FFF

For example, let's say we wanted to use block 13 (locations 832-895) and Sprite 1. We would do the following:

```
LDA     #13
STA     2041
```

That would do it. If you've worked with sprites in BASIC, it's just like POKE 2041,13.

Sprite Pointer

2. ENABLE Sprite

The register at $D015 (53269) is the ENABLE register for sprites. To enable a specific sprite, you load $D015 with the sprite *value,* not the sprite number.

Sprite Number	Sprite Value
Sprite 0	1
Sprite 1	2
Sprite 2	4

```
Sprite 3 ......   8
Sprite 4 ......  16
Sprite 5 ......  32
Sprite 6 ......  64
Sprite 7 ...... 128
```

To enable Sprite 2, you would do the following:

```
LDA     #4
STA     $D015
```

REMEMBER, the value 4 is for Sprite 2, not Sprite 4.

3. Store Sprite COLOR in Sprite Color Register.

Each sprite has a color value from 0-15 ($0-$F). The color values correspond to the standard colors we've used so far for background, border and character colors. Each sprite has its own color register beginning at $D027 (53287.) To determine which color register to use, just add the sprite number to the base address of $D027 (53287).

Sprite Number	Color Register
Sprite 0	$D027/53287
Sprite 1	$D028/53288
Sprite 2	$D029/53289
Sprite 3	$D02A/53290
Sprite 4	$D02B/53291
Sprite 5	$D02C/53292
Sprite 6	$D02D/53293
Sprite 7	$D02E/53294

For example, to store the color RED in Sprite 1 color register, simply use the following:

```
LDA     #2      ;Value for color RED
STA     $D028
```

The process works just like storing the background color in $D021 except it turns the sprite color on instead of the background color.

4. BUILD Sprite

At this stage, you load the sprite values (63 of them) into the block you stored in the pointer register in Step 1. For example, if you used Block 13, you would store the sprite values in locations 832-895 ($340-$37F). We will discuss the several ways sprites can be built and stored further on in this chapter.

5. Set horizontal HIGH bit to 0 or 1.

The register at $D010 holds the high byte for all sprite horizontal locations. If it is set to 1 then all horizontal (X) values are 256 or higher. If it is set to 0, all values are from 0-255. The horizontal screen for sprites is 320 dots wide. For the first 255 dots, the high byte is set to 0, and for 256 to 320, it must be set to 1. For the time being, we'll be setting it to 0.

```
LDA     #0
STA     $D010
```

Later we will discuss full horizontal movement.

6. MOVE Sprite

To move your sprite, the X (horizontal) and Y (vertical) positions are set in register pairs beginning at $D000-$D001 (53248-53249). The first of the pair is the X position and the second is the Y position.

	Register Pair	
Sprite Number	X	Y
Sprite 0	$D000/53248	$D001/53249
Sprite 1	$D002/53250	$D003/53251
Sprite 2	$D004/53252	$D005/53253
Sprite 3	$D006/53254	$D007/53255
Sprite 4	$D008/53256	$D009/53257

Sprite 5	$D00A/53258	$D00B/53259
Sprite 6	$D00C/53260	$D00D/53261
Sprite 7	$D00E/53262	$D00F/53263

To move a sprite, the X and Y values of the corresponding registers are changed. The good thing about sprites is that you do not have to erase a sprite to move it. You simply change the X and Y values of the corresponding registers. For example, to move Sprite 1 across the screen horizontally at vertical location 100, you would use the following:

```
        LDX     #0
        LDY     #100
        STY     $D003
MOVE    STX     $D002
        INX
        CPX     #255
        BNE     MOVE
```

Of course, you can also change the value of the Y register to move vertically, or change both the X and Y values to move diagonally.

At this point we have all of the basic information for making a sprite. For our first example, we will simply make a big block by filling in our sprite area with 255 ($FF). We will then move our block across the screen. Since machine language is so fast, we will have to slow our sprite down with a pause loop. We will use Sprite 0 for our first example since it uses the base addresses of all the registers. Also, we will use hexadecimal values for the registers since they are easier to remember. Most of them start with $D0, and you can think of them as "do" something or other.

GENERAL - SPRITE ZERO STARTER

```
LABEL           OPCODE  OPERAND COMMENT
--------------------------------------------------------------------------------
{Merlin         ORG     $8000}
{Commodore      * = $C000}
```

```
; ********************************************
; *                                          *
; *          SPRITE ZERO STARTER             *
; *                                          *
; ********************************************

{Commodore}
SPRITE0     =$7F8
ENABLE      =$D015
COLOR0      =$D027
SP0X        =$D000
SP0Y        =$D001
MSBX        =$D010
SHOUSE      =$0340

{Merlin}
SPRITE0     EQU     $7F8
ENABLE      EQU     $D015
COLOR0      EQU     $D027
SP0X        EQU     $D000
SP0Y        EQU     $D001
MSBX        EQU     $D010
SHOUSE      EQU     $0340
            JSR     $E544
            LDA     #$0D        ;Block 13 or
                                $0D
            STA     SPRITE0     ;Store it in
                                pointer for
                                Sprite 0
            LDA     #1          ;Sprite 0 enable
                                value
            STA     ENABLE      ;Store it in
                                enable register
            LDA     #2          ;Color red
            STA     COLOR0      ;Color register
                                for Sprite 0
            LDX     #0
            LDA     #0
CLEANUP     STA     SHOUSE,X
```

```
                                    ;Store zeros in
                                    sprite area
        INX
        CPX     #63
        BNE     CLEANUP
        LDX     #0
        LDA     #$FF
BUILD   STA     SHOUSE,X
                                    ;Fill up sprite
                                    area with $FF
                                    or 255
        INX                         ;to make solid
                                    block
        CPX     #63
        BNE     BUILD
        LDA     #0
        STA     MSBX                ;Store 0 in
                                    MSBX to locate
                                    sprite in
                                    horizontal
                                    locations 0 to
                                    255 only
        LDX     #0
        LDA     #70                 ;Vertical
                                    location
MOVE    STX     SP0X                ;Increment
                                    horizontal
                                    location by X
        STA     SP0Y                ;Y register stays
                                    at constant
                                    location of A
                                    register
        LDY     #0                  ;Delay loop.
PAUSE   INY
        CPY     #255
        BNE     PAUSE
        INX
        CPX     #254
        BNE     MOVE
        RTS
```

KIDS' ASSEMBLER - SPRITE-0 STARTER

```
ADRS     OPCODE  OPERAND
-------------------------------------------------------------------------------
49152    JSR     $E544
49155    LDA#    $0D
49157    STA     $7F8
49160    LDA#    1
49162    STA     $D015
49165    LDA#    2
49167    STA     $D027
49170    LDX#    0
49172    LDA#    0
49174    STA-X   $0340
49177    INX
49178    CPX#    63
49180    BNE     49174
49182    LDX#    0
49184    LDA#    $FF
49186    STA-X   $0340
49189    INX
49190    CPX#    63
49192    BNE     49186
49194    LDA#    0
49196    STA     $D010
49199    LDX#    0
49201    LDA#    70
49203    STX     $D000
49206    STA     $D001
49209    LDY#    0
49211    INY
49212    CPY#    255
49214    BNE     49211
49216    INX
49217    CPX#    254
49219    BNE     49203
49221    RTS
```

If you're using the Kids' Assembler, you're at a disadvantage since you cannot define the descriptive variable names. However,

with the Merlin and the Commodore assemblers, it's a good idea to create a source code with all of the sprite registers defined and then save it to disk. In this way, when you're ready to work with sprites, all of your variables are defined and you can start with the creation and movement of your sprites with little effort.

SPRITE BUILDING

The most exacting process in sprite programming is building the sprites. We'll design and build a sprite (something more interesting than a cross or block) and examine the different ways your assembler can store the sprite in memory. We will use Block 13 (832-895) in all of our examples. Instead of using the Kids' Assembler, we'll write a Sprite Assembler that will be a lot easier for creating and storing sprites. Our sprite will be called PLANET DESTROYER!

Making sprites takes
Patience...

PLANET DESTROYER

Column 1	Column 2	Column 3	Row
XXXXXXXX	XXXXXXXX	XXXXXXXX	1
XXXXXXXX	XXXXXXXX	XXXXXXXX	2
XXXXXXXX	XXXXXXXX	XXXXXXXX	3
XXXXXXXX	XXXXXXXX	XXXXXXXX	4
XXXXXXXX	XXXXXXXX	XXXXXXXX	5
+XXXXXXX	XXXXXXXX	XXXXXXXX	6
+++XXXXX	XXXXXXXX	XXXXXXXX	7
++++++XX	XXXXXXXX	XXXXXXXX	8
++++++++	+XXXXXXX	XXXXXXXX	9
++++++++	++++XXXX	XXX+++XX	10
XXX+++++	++++++++	++++XXXX	11
XX++++++	++++++++	++++++++	12

XX++++++	++++++++	++++++++	13
XXX+++++	++++++++	++++XXXX	14
++++++++	++++XXXX	XXXXXXXX	15
++++++++	+XXXXXXX	XXXXXXXX	16
++++++XX	XXXXXXXX	XXXXXXXX	17
+++XXXXX	XXXXXXXX	XXXXXXXX	18
+XXXXXXX	XXXXXXXX	XXXXXXXX	19
XXXXXXXX	XXXXXXXX	XXXXXXXX	20
XXXXXXXX	XXXXXXXX	XXXXXXXX	21

PLANET DESTROYER

Column 1	Column 2	Column 3	Row
00000000	00000000	00000000	1 0,0,0
00000000	00000000	00000000	2 0,0,0
00000000	00000000	00000000	3 0,0,0
00000000	00000000	00000000	4 0,0,0
00000000	00000000	00000000	5 0,0,0
100000000	00000000	0000000	6 128,0,0
1110000	00000000	00000000	7 224,0,0
11111100	00000000	00000000	8 252,0,0
11111111	10000000	00000000	9 255,128,0
11111111	11110000	00011100	10 255,240,28
00011111	11111111	11110000	11 31,255,240
00111111	11111111	00111111	12 63,255,63
00111111	11111111	00111111	13 63,255,63
00011111	11111111	11110000	14 31,255,240
11111111	11110000	00000000	15 255,240,0
11111111	10000000	00000000	16 255,128,0
11111100	00000000	00000000	17 252,0,0
11100000	00000000	00000000	18 224,0,0
100000000	00000000	0000000	19 128,0,0
00000000	00000000	00000000	20 0,0,0
00000000	00000000	00000000	21 0,0,0

At this point we have all the values for our PLANET DESTROYER sprite. The most obvious (and tedious) way of getting those values into memory would be to do something like the following:

```
LDA     #0
STA     832
LDA     #0
STA     833
etc...
```

With the Commodore and Merlin assemblers, a better way would be to define them with special pseudo-opcodes (directives). Remember when we used ASC to define strings? Well, you can also define numbers, except that you use DFB on Merlin and .BYTE on the Commodore EDITOR64. These directives work something like DATA statements. Each value is separated by a comma. For example, the following routine would read in the values in the line labeled DATA (any name will do).

```
                LDX     #0
BUILD           LDA     DATA,X
                STA     832,X
                INX
                CPX     #64
                BNE     BUILD

{Commodore}
DATA            .BYTE 0,0,0,0,0,0,0
                .BYTE 128,0,etc...

{Merlin}
DATA            DFB 0,0,0,0,0,0,0
                DFB 128,9,etc...
```

Now let's type in our PLANET DESTROYER program and give it a test run.

GENERAL - PLANET DESTROYER

```
LABEL           OPCODE  OPERANDCOMMENT
-------------------------------------------------------------------------------
{Merlin         ORG     $8000}
{Commodore      * = $C000}
```

```
;*********************************************
;*                                           *
;*                                           *
;*          PLANET DESTROYER                 *
;*                                           *
;*                                           *
;*********************************************

{Commodore}
SPRITE0         =$7F8
ENABLE          =$D015
COLOR0          =$D027
SP0X            =$D000
SP0Y            =$D001
MSBX            =$D010
SHOUSE          =$0340
{Merlin}
SPRITE0         EQU     $7F8
ENABLE          EQU     $D015
COLOR0          EQU     $D027
SP0X            EQU     $D000
SP0Y            EQU     $D001
MSBX            EQU     $D010
SHOUSE          EQU     $0340

                JSR     $E544
                LDA     #0          ;Color for black
                STA     $D020       ;Border black
                STA     $D021       ;Background
                                    black
                LDA     #$0D
                STA     SPRITE0
                LDA     #1
                STA     ENABLE
                LDA     #7          ;Yellow
                STA     COLOR0
                LDX     #0
                LDA     #$00
CLEANUP         STA     SHOUSE,X
                INX
                CPX     #64
```

```
                BNE     CLEANUP
                LDX     #0
BUILD           LDA     DATA,X      ;Read DFB /
                                    .BYTE values
                                    one at a time
                STA     SHOUSE,X
                                    ;Store in
                                    SpriteHOUSE.
                INX
                CPX     #63
                BNE     BUILD
                LDA     #0
                STA     MSBX
                LDX     #0
                LDA     #70
MOVE            STX     SP0X
                STA     SP0Y
                LDY     #0
PAUSE           INY
                CPY     #255
                BNE     PAUSE
                INX
                CPX     #254
                BNE     MOVE
                JMP     MOVE
                RTS
;Use .BYTE instead of DFB with Commodore

DATA            DFB     0,0,0,0,0,0,0,0,0
                DFB     0,0,0,0,0,0,128,0,0
                DFB     224,0,0,252,0,0
                DFB     255,128,0
                DFB     255,240,28
                DFB     31,255,240
                DFB     63,255,63
                DFB     63,255,63
                DFB     31,255,240
                DFB     255,240,0
                DFB     255,128,0
                DFB     252,0,0
```

```
DFB     224,0,0
DFB     128,0,0
DFB     0,0,0,0,0,0
```

Notice that when using DFB and .BYTE directives, the label, DATA, is able to reference all of the values entered. This includes values in lines after the label.

KIDS' SPRITE ASSEMBLER

As you can imagine, building sprites with the Kids's Assembler would be a royal pain in the neck since it has no labels or DFB / .BYTE directives. Therefore, let's make a special assembler that will make it easy to enter sprite data and save the sprite to disk. Then, once you have your sprite made, you can load it up into memory without having to build it each time you want to use it. In this way you can create a whole library of sprites, and then just write the programs to enable and move the sprites.

Since the Sprite Assembler is designed especially for sprites and not general assembly work, there are some special features about it. First, it takes data in groups of three. Each INPUT prompt expects three values separated by a comma. For example, you will be prompted,

```
ROW 5 ?
```

and you should enter something like,

```
ROW 5 ? 255,128,0 {RETURN}
```

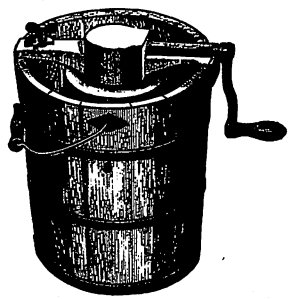
By using the method we have developed for calculating sprite values in the sprite matrix, it will be very simple to read sprites into memory.

When the program starts, you will be expected to give one of four blocks. The block number, not the beginning address is to be entered. When the program is saved to disk, the block number is appended to the file name so that later when you want to use that

sprite in a program, you will know the block number it uses. When you want the sprite in memory to be used by a program, you will load the sprite as follows:

```
LOAD "FILENAME 13",8,1
```

Then when the program accesses the data in Block 13, or whatever block it was saved in, it will find your sprite.



Sprite Assembler

Sprite Assembler

```
10 PRINT CHR$(147) : R=1
20 PRINT "STARTING ADDRESS AS BLOCK:"
30 PRINT "BLOCK 11: 704 – 767"
40 PRINT "BLOCK 13: 832 – 895"
50 PRINT "BLOCK 14: 896 – 959"
60 PRINT "BLOCK 15: 960 – 1023"
70 FOR X = 1 TO 39
80 PRINT CHR$(18);CHR$(32); : NEXT : PRINT
90 INPUT "BLOCK NUMBER";BL
100 IF BL <> 11 AND BL <> 13 AND BL <> 14
AND BL <> 15 THEN 90
110 SA = BL*64
120 FOR X=SA TO SA+62 STEP 3
130 PRINT "ROW";R
140 INPUT A,B,C
150 IF A > 255 OR B > 255 OR C > 255 THEN 130
160 POKE X,A : POKE X+1,B : POKE X+2,C
170 R = R + 1 : NEXT
200 REM **********
210 REM WRITE TO DISK
220 REM **********
230 LB = SA – INT(SA/256) * 256
```

```
240 HB = INT(SA/256)
250 INPUT "ENTER SPRITE NAME";SN$
260 SN$ = "0:" + SN$ + STR$(BL) + ",P,W"
270 OPEN2,8,2,SN$
280 PRINT#2,CHR$(LB) + CHR$(HB)
290 FOR V = SA TO SA + 62 : SC = PEEK(V)
300 PRINT#2,CHR$(SC)
310 NEXT V
320 CLOSE2
```

We should do the same thing for the other assemblers. After all, if we can save sprites as separate files, then we won't have to program around them. We can just load them into memory and then access them from any sprite program we write. The main thing to remember is that we have to use the available blocks of memory to store our sprites. Therefore the ORG must begin in one of those blocks. The following shows how to do this with Block 13. The same technique is used with the other blocks. Just change the ORG or * = address.

GENERAL - SPRITE CREATOR 13

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------------------------------------------
{Merlin         ORG     $0340}
{Commodore      * = $0340}

;*****************************************
;*                                       *
;*          SPRITE CREATOR 13            *
;*                                       *
;*****************************************

{Commodore}
SPRITE0         = $7F8
ENABLE          = $D015
COLOR0          = $D027
SP0X            = $D0
SP0Y            = $D001
MSBX            = $D010
SHOUSE          = $0340
```

```
{Merlin}
SPRITE0         EQU     $7F8
ENABLE          EQU     $D015
COLOR0          EQU     $D027
SP0X            EQU     $D000
SP0Y            EQU     $D01
MSBX            EQU     $D010
SHOUSE          EQU     $0340

;***********************************************
;*                                             *
;*                Build Sprite                 *
;*                                             *
;***********************************************

                LDX     #0
BUILD           LDA     DATA,X      ;Read DFB /
                                    .BYTE values
                                    one at a time
                STA     SHOUSE,X
                                    ;Store in
                                    SpriteHOUSE.
                INX
                CPX     #63
                BNE     BUILD


;***********************************************
;*                                             *
;*                Sprite Data                  *
;*                                             *
;***********************************************

{Merlin}
DATA            DFB     #,#,#       ;Values for
                                    sprite

{Commodore}
DATA            .BYTE   #,#,#
```

All you have to do is to enter your sprite data after the DFB or .BYTE directives. When you're finished, save the object code to disk, and then using either the LOADER64 programs on the Commodore or the LOAD "SPRITENAME.O",8,1 sequence for the Merlin object file.

= = THE WORM HAS TURNED = =

The Sprite Assembler is so convenient, you may actually want to use it instead of your regular assembler, especially if you're using the Commodore assembler. This is because you can use the LOAD "FILENAME",8,1 sequence to load sprites created with the Sprite Assembler. You can't do that with the Commodore system unless you create and save code with the monitor. So here's a case where the worm has turned and it's actually easier to use something from the Kids' system.

Now that we have programs that will create sprites and save them for use, we will need to see how to access these sprites from within a program. We'll just modify our starter program to be a Sprite Tester. We can load a sprite from disk, and then run the following program to see if the sprite is what we thought it was. Notice that we have taken out the CLEANUP subroutine. This is because that would wipe out any sprite in memory we wanted to test. This tester is designed for sprites using Block 13, but it can be changed to test sprites in any block. Use the following sequence:

1. Create your sprite and save it to disk.
2. Load your sprite into memory as you would any other machine language program.
3. Load the SPRITE TESTER program into memory and SYS it.

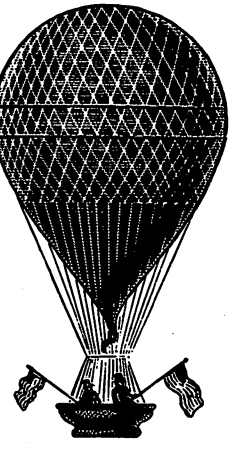
Your sprite should run across the screen for you.

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------------------------------------------
{Merlin         ORG     $8000}
{Commodore      * = $C000}

; ***********************************************
;*                                              *
;*                SPRITE TESTER                 *
;*                                              *
; ***********************************************

{Comm0odore}
SPRITE0         = $7F8
ENABLE          = $D015
COLOR0          = $D027
SP0X            = $D000
SP0Y            = $D001
MSBX            = $D010
SHOUSE          = $0340
{Merlin}
SPRITE0         EQU     $7F8
ENABLE          EQU     $D015
COLOR0          EQU     $D027
SP0X            EQU     $D000
SP0Y            EQU     $D001
MSBX            EQU     $D010
SHOUSE          EQU     $0340
                JSR     $E544
                LDA     #0
                STA     $D020
                STA     $D021
                LDA     #$0D
                STA     SPRITE0
                LDA     #1
                STA     ENABLE
                LDA     #7
                STA     COLOR0
                LDX     #0
```

```
            LDA     #$00
            STA     MSBX
            LDA     #70
MOVE        STX     SP0X
            STA     SP0Y
            LDY     #0
PAUSE       INY
            CPY     #255
            BNE     PAUSE
            INX
            CPX     #254
            BNE     MOVE
            RTS
```

KIDS' ASSEMBLER - SPRITE TESTER

```
ADRS      OPCODE  OPERAND
-------------------------------------------------------------------------------
49152     JSR     $E544
49155     LDA#    0
49157     STA     $D021
49160     STA     $D020
49163     LDA#    $0D
49165 STA $7F8
49168     LDA#    1
49170     STA     $D015
49173     LDA#    7
49175     STA     $D027
49178     LDX#    0
49180     LDA#    0
49182     STA     $D010
49185     LDA#    70
49187     STX     $D000
49190     STA     $D001
49193     LDY#    0
49195     INY
49196     CPY#    255
49198     BNE     49195
49200     INX
49201     CPX#    254
```



Testing Sprite

```
49203   BNE     49187
49205   RTS
```

FULL HORIZONTAL MOVEMENT

So far, we have only moved part of the full horizontal move. In order to move our full horizontal width, we have to store a 1 in the MSBX register. That's easy enough. Just insert the following lines to the SPRITE TESTER program before the RTS and your sprite will move across the entire screen:

```
        LDX     #0
        LDA     #1
        STA     MSBX    ;Set the high
                        byte of X
                        position
        LDA     #70     ;Keep Y at 70
MOVE2   STX     SP0X    ;X position is
                        now 255 + X
        STA     SP0Y
        LDY     #0
PAUSE2  INY
        CPY     #255
        BNE     PAUSE2
        INX
        CPX     #90
        BNE     MOVE2
```

When you test a sprite on the program now, it will go all the way across the screen.

SPRITE EXPANSION

A final aspect of sprite programming we will discuss is the expansion of sprites to double size. The registers at $D01D (53277) and $D017 (53271) control the X and Y sizes of the sprite. The sprites we have been using so far have been single width and height. By storing the Sprite Value in the respective registers, it is possible to expand the sprite to double width. The following sprite values are used:

Sprite	Value
Sprite Ø	1
Sprite 1	2
Sprite 2	4
Sprite 3	8
Sprite 4	16
Sprite 5	32
Sprite 6	64
Sprite 7	128

It is fairly important to have a variable name for your X and Y expansion registers. This is so that you can add and subtract the sprite value to those registers to change the sprite size. If you're using multiple sprites, this becomes crucial. For example, if you store 1 in $DØ17 and $DØ1D, Sprite Ø will be expanded. If you then STA 2 in the same registers so that Sprite 1 will be expanded, Sprite Ø will be reset to normal size. The following shows how to change sprite size using multiple sprites.

```
YXPAND  EQU     $DØ17
XXPAND  EQU     $DØ1D
        LDA     #1          ;Sprite Ø value stored
                            in expansion
                            registers
        STA     YXPAND
        STA     XXPAND
        LDA     YXPAND
        CLC
        ADC     #2          ;Add Sprite 1 value to
                            registers
        STA     YXPAND
        STA     XXPAND
```

Now if you want to return one or the other sprites to normal size, you simply subtract from the expansion registers. For example, to keep Sprite 1 at double size and return Sprite Ø to normal, you would just SBC 1 from each register.

```
LDA     XXPAND
SEC
SBC     #1
STA     YXPAND
STA     XXPAND
```

There is still more to sprites than we have covered, but we have dealt with the major aspects of assembly language programming and sprites. The trick is to get organized with sprites. Then, it is simply a matter of following a series of steps. By using the keyboard and joystick subroutines we've developed, it is possible to control movement from these external devices. Since you should already know how to do that, I'll let you insert these subroutines yourself.

ASSEMBLY SOUNDS

Like sprite programming in BASIC, most sound programming is also done with POKEs and PEEKs. Therefore, it should not be at all difficult to do make all the sounds we want. The trick, as with sprites, is to organize everything into variables pertaining to the proper registers.

We're going to look at the basic features of sound on your Commodore 64. By making changes to the program we're going to write, you will be able to produce a whole lot of sound. By changing the variables, you will be able to make music, racket or the sound of a train rushing through your TV. Here, the trick is to see how to get your sounds working in assembly language. Other sources are available for getting all of the several aspects of your SID chip working. Just in case you have not worked with sound before, turn up the sound on your TV so you can hear what's going on. (If you have a monitor with no sound, you might as well skip this section.)

On the most fundamental level, you computer deals with six registers to create sound. The duration of a sound is done with a delay loop that keeps everything going until the program nulls the sound registers by filling them with zeros.

1. Volume Control. The volume control register is at $D418 (54296). It can have a maximum value of $F (15). It controls how loud the sound will be.

2. Attack - Decay. This refers to how fast a sound reaches its maximum volume and falls from that volume. Its register is at $D405 (54277). The maximum value is 240.

3. Sustain - Release. The extent to which a sound is carried at a certain level before it is released is its sustain/release variable. The $D406 (54278) register holds it. Like attack/decay, its maximum value can be 240.

4. High and Low Frequency. The pitch of notes is determined by their high/low frequency. The low frequency register is at $D400and the height at $D401.

5. Waveform. Sounds can either be smooth or rough. The waveform determines how this will occur. There are four major waveforms we can use:

a. Triangle = 17
b. Sawtooth = 33
c. Pulse = 65
d. Noise = 129

To make a sound, we first store values in the registers. Then we simply "hold" a sound with a delay loop. Even a loop of 255 will give you a very short sound; so you may need nested loops, depending on what sound you want. At the end of the delay loop, you will null the registers by storing zeros in them.

GENERAL - ASSEMBLY SOUND

```
LABEL          OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin        ORG     $8000}
{Commodore     * = $C000}
```

```
;*********************************************
;*                                           *
;*          ASSEMBLY SOUNDS                  *
;*                                           *
;*********************************************

{Commodore}
SIGVOL      =$D418
ATDCY1      =$D405
SUREL1      =$D406
VCREG1      =$D404
FRELO1      =$D400
FREHI1      =$D401
{Merlin}
SIGVOL      EQU     $D418
ATDCY1      EQU     $D405
SUREL1      EQU     $D406
VCREG1      EQU     $D404
FRELO1      EQU     $D400
FREHI1      EQU     $D401
            LDA     #15
            STA     SIGVOL    ;Set volume to
                              15
            LDA     #128
            STA     ATDCY1    ;Set attack
                              - decay to 128
            STA     SUREL1    ;Set sustain
                              - release to 128
            LDA     #195
            STA     FRELO1    ;Store 195 in the
                              low frequency
            LDA     #16
            STA     FREHI1    ;Store 16 in the
                              high frequency
            LDA     #17
            STA     VCREG1
            LDY     #0
SET         LDX     #0
PLAY        INX
```

```
CPX     #255     ;Double delay
                 loop to play
                 sound
BNE     PLAY
INY
CPY     #100
BNE     SET
LDA     #0       ;Null registers
                 with 0
STA     VCREG1
STA     ATDCY1
STA     SUREL1
STA     FRELO1
STA     FREHI1
RTS
```

KIDS' ASSEMBLER - ASSEMBLY SOUNDS

ADRS	OPCODE	OPERAND
49152	LDA#	15
49154	STA	$D418
49157	LDA#	128
49159	STA	$D405
49162	STA	$D406
49165	LDA#	195
49167	STA	$D400
49170	LDA#	16
49172	STA	$D401
49175	LDA#	17
49177	STA	$D404
49180	LDY#	0
49182	LDX#	0
49184	INX	
49185	CPX#	255
49187	BNE	49184
49189	INY	
49190	CPY#	100
49192	BNE	49182
49194	LDA#	0

```
49196   STA   $D404
49199   STA   $D405
49202   STA   $D406
49205   STA   $D400
49208   STA   $D401
49211   RTS
```

You should try changing everything from the size of the delay loop to the values stored in the registers. By doing so, you will discover the sounds you need and hear ones that will surprise you. Combine the sounds with your sprites and you'll have the makings of an arcade game!

SUMMARY

This chapter is important in that we used two special chips and sets of registers in your Commodore 64. With the VIC chip, we were able to access sprite graphics. This allowed us to produce and move special characters we can create ourselves. Secondly, we used the registers in the SID chip to create sound. The combination of these two sets of registers in single programs will let you do some very interesting things. In the next chapter, we'll look at some examples.

CHAPTER 15

DOWN THE ROAD

Introduction

This last chapter is to help you go on in assembly language once you're finished here. Like any other language, programming or spoken, the real secret is to use it. If you study German in school, for example, and you never use it, you'll forget it. On the other hand, if you live in Germany and are constantly speaking German, soon you will become fluent. The same is true with assembly language. When you sit down to write a program, even in BASIC, ask yourself, "How could the same thing be done in assembly language?" You can start off with little subroutines to speed up your BASIC programs and work your way up to full blown programs, all written with an assembler.

Practice will make perfect, but you still have a lot of new opcodes to learn with which you can practice. We will look at some tricks to help you learn new opcodes.

Finally, you're going to need more materials to study to learn assembly language. For the Commodore 64, there are some available materials that are highly recommended. We'll look at those so that you can keep progressing beyond this point.

MERGING SUBROUTINES

Assembly language programming is best digested as a 50 course dinner. If you take just a little at a time, you will be able to consume everything. These little bites are best conceived as "subroutines." This actually involves writing little programs that do something as we have done in this book. Each little program, when merged with other little programs, becomes a big program.

To see how to append programs, we will have to use the Merlin and Commodore assemblers as examples since they both can append programs and the Kids' Assembler cannot. That means, that you can load one source code into the editor and then tack on another source code to it.

Appending With Merlin

To append one source code to another, first load one source code using the "L" instruction from the EXECUTIVE MODE. Once the first file is loaded return to the EXECUTIVE MODE by pressing 'Q' from the editor, and use the "A" option to append the second file.

Appending with Commodore EDITOR64

Once the editor is in memory, use the GET command to load the first file. All files loaded with GET begin at line 1000. List your program to see the highest line number. For example, let's say that the highest line number in your program is 1200. To append a program to the one in memory, use the GET command with the parameter for the first line number. That line number must be higher than 1200. For example, we'll append FILE 2 to FILE 1.

```
1. GET "FILE 1" {RETURN}
2. LIST {RETURN}
   Highest line number is 1200
3. GET "FILE 2",1210
```

Now FILE 2 would be appended to FILE 1.

To do something new with the programs we have developed in this book, let's append some. In the last chapter we made a "Planet Destroyer" sprite and "Assembly Sounds." Let's put those two together, edit them and see what we get. Load "Planet Destroyer" and then append "Assembly Sounds." We'll look at the program first and then explain what editing changes were made to make it. (We'll use the Merlin listing since the only difference is in the formatting of EQUates, and with the large number of variables defined, having two sets would be confusing.)

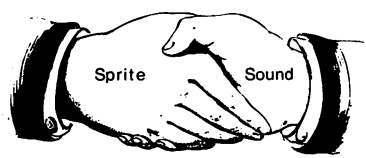
GENERAL SLOW NOISY SPACE SPRITE

```
LABEL           OPCODE  OPERAND COMMENT
----------------------------------------------------------------------------
{Merlin         ORG     $C000}
{Commodore      * = $C000}
*************************************************
*                                               *
*              NOISY SPRITE ROCKET              *
*                                               *
*************************************************

SPRITE0         EQU     $7F8
ENABLE          EQU     $D015
COLOR0          EQU     $D027
SP0X            EQU     $D000
SP0Y            EQU     $D001
MSBX            EQU     $D010
SHOUSE          EQU     $0340
SIGVOL          EQU     $D418
ATDCY1          EQU     $D405
SUREL1          EQU     $D406
VCREG1          EQU     $D404
FRELO1          EQU     $D400
FREHI1          EQU     $D401
HI              EQU     $C300
WHY             EQU     $C304
EX              EQU     $C306
```

```
;SPRITE SUBROUTINE
; ***********************************************
                JSR     $E544
                LDA     #$0D
                STA     SPRITE0
                LDA     #1
                STA     ENABLE
                LDA     #2
                STA     COLOR0
                LDX     #0
                LDA     #$00
CLEANUP         STA     SHOUSE,X
                INX
                CPX     #64
                BNE     CLEANUP
                LDX     #0
BUILD           LDA     DATA,X
                STA     SHOUSE,X
                INX
                CPX     #63
                BNE     BUILD
                LDA     #0
                STA     MSBX
                LDX     #0
                STX     EX
                LDA     #70
                STA     WHY         ;Preserve "Y"
MOVE            STX     SP0X
                LDA     WHY
                STA     SP0Y
                INC     EX          ;Increment X
                                    through EX
                LDX     EX
;SOUND SUBROUTINE
;NO PAUSE LOOP SINCE THE SOUND SLOWS
;DOWN MOVEMENT
; ***********************************************

                LDA     #10
                STA     HI
```

```
START       LDA     #15
            STA     SIGVOL
            LDA     #128
            STA     ATDCY1
            STA     SUREL1
            LDA     HI
            STA     FRELO1
            LDA     HI
            STA     FREHI1
            LDA     #17
            STA     VCREG1
            LDY     #0
SET         LDX     #0
PLAY        INX
            CPX     #20
            BNE     PLAY
            INY
            CPY     #20
            BNE     SET
            LDA     #0
            STA     VCREG1
            STA     ATDCY1
            STA     SUREL1
            STA     FRELO1
            STA     FREHI1
            INC     HI
            LDA     HI
            CMP     #40
            BNE     START
            LDX     EX
            CPX     #254
            BNE     MOVE      ;Back to Sprite

;END OF PROGRAM AND SPRITE DATA
*********************************************
            RTS
DATA                  DFB 0,0,0,0,0,0,0,0,0
            DFB       0,0,0,0,0,0,128,0,0
            DFB       224,0,0,252,0,0
            DFB       255,128,0
```

```
DFB     255,240,28
DFB     31,255,240
DFB     63,255,63
DFB     63,255,63
DFB     31,255,240
DFB     255,240,0
DFB     255,128,0
DFB     252,0,0
DFB     224,0,0
DFB     128,0,0
DFB     0,0,0,0,0,0
```



If you look at the sprite data, it's the same we used with the Planet Destroyer sprite. You can change it to any sprite configuration you want, and using your append features of your assembler, it is easy. The main changes we made were in the form of substitution. Instead of using the INX instruction to increment the horizontal position of the sprite, we stored the X position in the 'variable' (location) we called EX and incremented the value in EX. We also used a variable, WHY, to store the vertical position of the sprite. The reason we did that is because the X and Y registers were heavily used in producing our sound. Since the sound values were not directly related to the movement values of X and Y, we needed some way to preserve the X and Y values in movement. By only using the values in EX and WHY in the movement segment and resetting the X and Y registers in the sound portion of the program, we were able to keep everything moving and sounding the way we intended.

While the actual movement mechanism in not tied to the sound, the speed of the movement is. If the various sizes of the loops in the sound segment of the program are shortened, the sound will change and so will the speed of the rocket. The way it is now, it makes a "space gurgle" and creeps along at a snail's pace. (Actually, it's sneaking up on a planet to destroy.)

Finally, you probably noticed that all of the EQUates are together at the beginning of the program. This was done with the MOVE function on Merlin. With the Commodore 64 Macro Assembler Development System, you can do the same thing by changing the line numbers of the equate (=) directives. Then, delete the old line numbers and everything is moved.

APPENDING AND INSERTING SUBROUTINES

In the above program, we essentially stuck two programs together; one on top of the other. Now, we'll do something a little more complex. We'll start the same way, though. First, load "Planet Destroyer" and then append "Joystick Draw." What we're going to do is to insert the joystick read subroutine inside the sprite routine. Take out the routine to increment the X register to move, and in its place put the joystick read subroutine. Then, change the JMPs to PRINT to JMPs to MOVE. Both the Merlin and Commodore editors have CHANGE commands you can use to do this quickly.

We added another wrinkle to our joystick program. When the values in EX and WHY reach the limits of Ø and 255, we don't want them to "turn over." Instead, we just have them jump to the routine that will increment them or decrement them so they stay within the registers' boundaries. Thus, when UP or LEFT is Ø, the program will JMP to DOWN or RIGHT. Similarly, when DOWN or RIGHT reaches 255, the value is decremented by JMPs to UP and LEFT.

```
LABEL           OPCODE  OPERAND COMMENT
-------------------------------------------------------------------------------
{Merlin         ORG     $CØØØ}
{Commodore      * = $CØØØ}

*************************************************
*                                               *
*              JOYSTICK SPRITE                  *
*                                               *
*************************************************
```

```
CLEAR       EQU     $E544
JSTICK      EQU             $DC01
OFSET       EQU     $C300
FIRE        EQU     $C302
EX          EQU     $C304
WHY         EQU     $C306
SPRITE0     EQU     $7F8
ENABLE      EQU     $D015
COLOR0      EQU     $D027
SP0X        EQU     $D000
SP0Y        EQU     $D001
MSBX        EQU     $D010
SHOUSE      EQU     $0340

; ***********************************************

            JSR     CLEAR
            LDA     #$0D
            STA     SPRITE0
            LDA     #1
            STA     ENABLE
            LDA     #2
            STA     COLOR0
            LDX     #0
            LDA     #$00
CLEANUP     STA     SHOUSE,X
            INX
            CPX     #64
            BNE     CLEANUP
            LDX     #0
BUILD       LDA     DATA,X
            STA     SHOUSE,X
            INX
            CPX     #63
            BNE     BUILD
            LDA     #0
            STA     MSBX
            LDA     #$FF
            STA     OFSET
            LDA     #16
```

```
                STA     FIRE
                LDA     #40
                STA     EX      ;Starting X
                                position of
                                Sprite
                STA     SP0X
                LDA     #70
                STA     WHY     ;Starting Y
                                position of
                                Sprite
                STA     SP0Y

;READ THE JOYSTICK
; ***********************************************

START           LDA     JSTICK
                EOR     OFSET
                CMP     #1
                BEQ     UP
                CMP     #2
                BEQ     DOWN
                CMP     #4
                BEQ     LEFT
                CMP     #8
                BEQ     RIGHT
                CMP     FIRE
                BEQ     END
                LDX     #0
PAUSE           INX

                CPX     #254
                BNE     PAUSE
                JMP     START
UP              DEC     WHY
                LDY     WHY
                CPY     #0      ;Check for
                                adjustment
                BEQ     DOWN
```

```
                JMP     MOVE    ;Branch to
                                MOVE
                                subroutine
DOWN            INC     WHY
                LDY     WHY
                CPY     #255
                BEQ     UP
                JMP     MOVE
LEFT            DEC     EX
                LDX     EX
                CPX     #0
                BEQ     RIGHT
                JMP     MOVE
RIGHT           INC     EX
                LDX     EX
                CPX     #255
                BEQ     LEFT
                JMP     MOVE
END             RTS

;MOVEMENT OF SPRITE
; ********************************************
;
MOVE            STX     SP0X
                STY     SP0Y
                LDX     #0
MPAUSE          INX
                CPX     #254
                BNE     MPAUSE
                JMP     START

;SPRITE DATA
; ********************************************
;
DATA            DFB     0,0,0,0,0,0,0,0,0
                DFB     0,0,0,0,0,0,128,0,0
                DFB     224,0,0,252,0,0
                DFB     255,128,0
                DFB     255,240,28
                DFB     31,255,240
                DFB     63,255,63
```

```
DFB     63,255,63
DFB     31,255,240
DFB     255,240,0
DFB     255,128,0
DFB     252,0,0
DFB     224,0,0
DFB     126,0,0
DFB     0,0,0,0,0,0
```

These two programs should give you an idea of how to merge two smaller routines into a single larger one. Now you can treat the larger routines as subroutines themselves. You have a subroutine that will move your sprites and create sound and one that will move sprites with your joystick. By appending different sprite data, adding additional sprites, and appending other subroutines you can build larger programs. The trick is to build a library of subroutines, and instead of starting from scratch every time you sit down to program, you can start with a substantial amount of the work already done and debugged. This also points to the importance of getting a good assembler. While the Kids' Assembler is fine for learning assembly language programming and small programs, it becomes increasingly difficult to use as your programs increase. Starting all over each time you program is a waste of time, and in the long run you will not learn as much since more effort is used keying in old code than creating new code.

GETTING TO KNOW THE OTHER OPCODES

We've covered the most commonly used opcodes and addressing modes, but there are still more to learn and use. The best way to learn how to use new opcodes is to first isolate them, and then test them with simple programs. Often novice assembly language programmers will insert a new opcode into a large program, and while it may or may not work, it is difficult to see why it did or did not.

Usually the best way to see if an opcode does what you think it does is to put it into a routine that will put a value into the accumulator. Then have that value sent to the screen. If the screen character is what you expect; then you can assume that the opcode did what you thought. In the listing in Appendix A, the Kids'

Assembler there has all of the opcodes for the 6510, and Appendix B has an alphabetical listing of all 6510 opcodes.

A good book on 6502 opcodes (which are identical to the 6510opcodes) is:

6502 Assembly Language Programming
By Lance A. Leventhal
Berkeley, CA : Osborne/McGraw-Hill

This book describes in technical detail what each code does.

RESOURCES FOR LEARNING MORE ABOUT ASSEMBLY LANGUAGE PROGRAMMING ON THE COMMODORE 64

As more good assemblers are becoming available for the Commodore 64, there should be more resources for learning assembly language programming. Most assembler manuals have very little about programming. They usually just explain how to use the assembler, more or less assuming you already know how to program. However, from what you know now, you should be able to understand what the assembler manuals are talking about and use them to some advantage.

Since there are a wide variety of resources for learning assembly language, we'll divide them into several categories. First, we'll look at reference manuals. These are books used to look things up. Secondly, we'll see what other books there are on "how to" do assembly language programming. Third, we'll look at magazines you might read to either learn assembly language programming or see some examples of assembly language programs. Before all of that, though, we'll look at the best resource possible, Commodore 64 User Groups.

USER GROUPS

The most valuable tips, techniques and education you can receive for assembly language on the Commodore 64 is from a user group. These are people who own Commodore 64's and get together to share their common interest. These groups typically have several

members who are kids like yourself. (My group even has a lot of adults who are just like kids.) If you attend meetings in your area, not only will you meet other kids who have Commodore 64's, you can learn a lot about assembly language programming by just asking.

Now you may wonder why people are going to give you all of this free information. It's simple; assembly language programmers love to brag. I do it all the time. If someone asks about an assembly language procedure, nothing makes me feel better than being able to answer their question. If you know BASIC, you've probably experienced the same thing. You can help someone by showing off! What could be better? Not only will you learn something, you'll make someone else feel very good about themselves.

In addition to getting a wealth of information, you can also gain access to public domain (that means FREE!) software. Since a good deal of public domain software is written in assembly language, you can use assembly listing to see how others write assembly code on the Commodore 64.

REFERENCE BOOKS

There are two reference books no assembly language programmer should be without. First, there is;

Commodore 64 Programmer's Reference Guide
Commodore Business Machines, Inc.
And Howard W. Sams & Co., Inc.

This book has all of the technical details you will need to work your Commodore 64. Most important is its description of the Kernal routines in your machine.

Secondly, your assembly language programming will be greatly enhanced by the book,

Mapping the Commodore 64
By Sheldon Leemon
Greensboro, N.C. : COMPUTE! Publications, Inc., 1984

All of the built-in subroutines in your Commodore 64, along with substantial explanations of their use are contained in this book. It is well written, and there are several example programs in BASIC that illustrate how the various subroutines work.

HOW-TO BOOKS

There really are not a lot of books available for assembly language programming on the Commodore 64 right now. However, there are three books you might want to examine for the transition from BASIC to machine/assembly language.

The Intermediate Commodore 64
By Guy Grotke
Chatsworth, CA : Datamost, 1984

This book builds a bridge between BASIC and assembly language programming. Some of the more advanced aspects of graphics, files and basic assembly language procedures are explained. There are a number of listings of assembly language routines and the appendix has all of the decimal and hexadecimal values for the 651Ø opcodes.

Commodore 64 Exposed
By Bruce Bayley
Melbourne, Australia: Melbourne House, 1983

Like the Intermediate Commodore 64, this is a "transition book" between BASIC and more advanced techniques. There are some assembly language listings, a good memory map and lots of good tips.

Inside the Commodore 64
By Don French
Cannon Falls, MN : French Silk

This is actually the manual that accompanies the French Silk Assembler. More than most manuals, this one goes beyond just explaining how to use the assembler. There are several example pro-

grams, an excellent memory map, and you can get it with the assembler or without.

MAGAZINES

The problem with most magazines for the Commodore 64 is that they create machine code with BASIC programs. As a result, their listings are in the form of READ/DATA statements that are POK-Ed into memory. You can't see the assembly opcodes, but rather you just get a pile of decimal machine codes. However, there are some assembly listings, and you can still learn something about machine language programs even if the listings are given in BASIC. With Merlin's Sourceror, you can create source code from the machine code generated with the BASIC programs in the magazines. Thus, with the right tools, you can extend your knowledge a good deal.

The most likely source for actual assembly listings is in *Commander* magazine. Like most other magazines, this one will have machine code generated by BASIC programs, but it also has disassembled listings as well. Previous issues have contained series articles, such as "Explorations With Assembly Language" by Eric Giguere. You'll find plenty of new material just about every month in the *Commander.*

A second source for machine code is in *COMPUTE!'s Gazette.* There's a regular series, "Machine Language For Beginners" by Richard Mansfield you will find useful. In previous articles, there have been assembly listings for various aspects of machine language techniques. Mansfield handles the material in small, clear chunks, and you can get a lot here.

Third, *RUN: The Commodore 64 & VIC 20 Magazine,* has a number of programs and articles dealing with machine and assembly language programming. Several very sophisticated programs can be found in this publication.

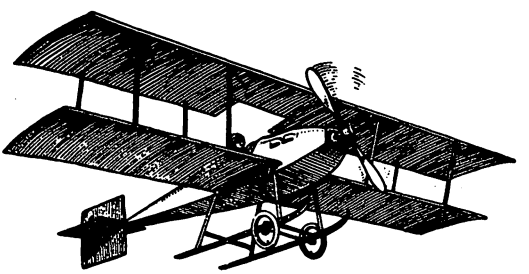
Other magazines for the Commodore 64 are available as well. The best idea is to take a look at the various issues, including the three recommended above, and see if there are any articles or pro-

grams on assembly/machine language programming. An even better idea would be to write a letter to the editor of your favorite Commodore 64 magazine and tell them you want to see more listings and articles on assembly language programming.

YOU'RE ON YOUR OWN

By this stage, you're well on your way to becoming a full-fledged assembly language programmer. If you've gotten this far, you're way ahead of where you were at the beginning. You should be able to write routines of your own creation with the more common opcodes and most addressing modes. What you do next is up to you.

If nothing else, I hope to have conveyed the fundamentals of assembly language programming and given you enough confidence to experiment on your own. If you can do that; then you'll be able to go much further. Above all, assembly language programming should be an exciting challenge. Put another way, it should be plain fun. Before you know it, you'll wonder why anyone programs in BASIC.



Now you can **FLY.**

APPENDICES

APPENDIX A: KIDS' ASSEMBLER
APPENDIX B: 6510 OPCODES
APPENDIX C: MEMORY MAP 1: DIAGRAM
APPENDIX D: MEMORY MAP 2: PLACES TO VISIT
APPENDIX E: BASIC TOKEN CHART
APPENDIX F: HEXADECIMAL-DECIMAL CONVERSION CHART
APPENDIX G: DECIMAL-HEXADECIMAL CONVERSION CHART
APPENDIX H: SCREEN STORAGE ADDRESS TABLE
APPENDIX I: COLOR STORAGE LOCATION TABLE
APPENDIX J: ASCII CODE
APPENDIX K: SCREEN STORAGE DISPLAY CODES

APPENDIX A

KIDS' ASSEMBLER

This version of the Kids' Assembler has all opcodes for the 6510. There are two ENDING ROUTINEs, one for disk and one for tape. The disk version is in the main listing between lines 740 and 960. If you are using a cassette, there is a second ENDING ROUTINE at the end of the main listing.

```
10 POKE 53281,1 : POKE 53280,1 : PRINTCHR$(144)
20 PRINT CHR$(147)
30 DIM DEC%(151),OPCODE$(151),BYTE%(151)
40 GOSUB 2510
50 FOR I = 0 TO 150 : READ DEC%(I) : READ
OPCODE$(I) : READ BYTE%(I)
60 NEXT I
70 PRINT CHR$(146);CHR$(147)
80 PRINT "ADRS";
TAB(10);"OPCODE";TAB(25);"OPERAND"
90 FOR X = 1 TO 40 : PRINT CHR$(114); : NEXT
100 PRINT
110 REM ****************************
120 REM SET ADDRESS AND INPUT OPCODE
```

```
130 REM ****************************
140  SA=0  :  PRINT  "PRESS  {RETURN}
 TO DEFAULT TO
49152"
150 INPUT"STARTING ADDR";SA : IF SA=0
THEN SA=49152
160 BA=SA
170 PRINT SA;TAB(10)180 INPUT OC$ : IF OC$="Q"
THEN 740
190 C=0
200 IF OC$=OPCODE$(C) THEN D%=DEC%(C) :
B%=BYTE%(C) : GOTO 230
210 C=C+1 : IF C=152 THEN PRINT
TAB(10);CHR$(18);"ERROR";CHR$(146) : GOTO 170
220 GOTO 200
230 IF B%=1 THEN POKE SA,D%: SA=SA+1 :
GOTO 170
240 REM ************
250 REM ENTER OPERAND
260 REM ************
270 PRINT TAB(25);:PRINT CHR$(145);:INPUT OPR$
280 IF LEFT$(OPR$,1) <> "$" THEN
OPER=VAL(OPR$)
290 IF LEFT$(OPR$,1)="$" THEN GOSUB 450
300 IF OPER > 65535 THEN GOSUB 590 : OPER=0:
GOTO 270
310 IF OC$="BNE" OR OC$="BEQ"
THEN GOSUB 660
320 IF OC$="BCC" OR OC$="BCS"
THEN GOSUB 660
330 IF OC$="BPL" OR OC$="BMI"
THEN GOSUB 660
340 IF OC$="BVC" OR OC$="BVS"
THEN GOSUB 660
350 IF BF=1 THEN BF=0 : GOTO 270
360 IF OPER > 255 AND B% < 3 THEN GOSUB 520:
OPER=0 : GOTO 270
370 IF OPER > 255 THEN GOSUB 600
380 REM ************
390 REM COMPILE CODE
```

```
400 REM ************
410 IF B% = 2 THEN POKE SA,D% : SA = SA + 1
420 IF B% = 2 THEN POKE SA,OPER : SA = SA + 1 :
OPER = 0 : GOTO 170
430 POKE SA,D%: SA = SA + 1
440 POKE SA,LB : SA = SA + 1 : POKE SA,HB :
SA = SA + 1 : OPER = 0 : GOTO 170
450 REM ***********************
460 REM CONVERT HEX TO DECIMAL
470 REM ***********************
480 H$ = MID$(OPER$,2)
490 FOR L = 1 TO LEN(H$) : HD = ASC(MID$(H$,L,1))
500 OPER = OPER*16 + HD – 48 + ((HD > 57)*7)
510 NEXT L : RETURN
520 REM **********
530 REM ERROR TRAP
540 REM **********
550 PRINT CHR$(18);"ERROR – MUST BE LESS
THAN 256"
560 FOR W = 1 TO 400 : NEXT W : PRINT CHR$(146); :
PRINT CHR$(145)570 FOR X = 1 TO 27 : PRINT
CHR$(32); : NEXT
580 PRINT CHR$(157);CHR$(157);CHR$(145) :RETURN
590 PRINT CHR$(18);"VALUE OVER 65535
($FFFF)";CHR$(146) : RETURN
600 REM ************************
610 REM CONVERT TO 2 BYTE NUMBER
620 REM ************************
630 LB = OPER – INT(OPER/256)*256
640 HB = INT(OPER/256)
650 RETURN
660 REM *************
670 REM BRANCH OFFSET
680 REM *************
690 IF SA > OPER AND SA – OPER > 128 THEN
PRINT "BRANCH TOO
FAR":BF = 1:OPER = 0:RETURN
700 IF SA > OPER AND OPER – SA > 127 THEN
PRINT "BRANCH TOO
FAR":BF = 1:OPER = 0:RETURN
```

```
710 IF SA  > OPER THEN OPER= 254 – (SA – OPER)
720 IF SA  < OPER THEN OPER= (OPER – SA) – 2
730 RETURN
740 REM **************
750 REM ENDING ROUTINE
760 REM **************
770 NB  = SA – BA
780 PRINT CHR$(147)
790 FOR X = 1 TO 5 : PRINT : NEXT
800 INPUT"SAVE PROGRAM(Y/N)";AN$
810 IF AN$ = "Y" THEN 870
820 PRINT : PRINT : PRINT "PROGRAM
IS";NB;"BYTES LONG"
830 PRINT "TO EXECUTE 'SYS' ";BA : PRINT
840 INPUT "(B)EGIN AGAIN OR (E)ND";DE$
850 IF DE$ = "B" THEN 70
860 PRINT : PRINT"END" : END
870 PRINT CHR$(147) : FOR X = 1 TO 5 : PRINT :
NEXT
880 LB = BA – INT(BA/256)*256 : HB = INT(BA/256)
890 INPUT "ENTER FILE NAME";NF$
:NF$ = "0:" + NF$ + STR$(BA) + ",P,W"
900 OPEN2,8,2,NF$
910 PRINT#2,CHR$(LB) + CHR$(HB)920 FOR X = BA
TO SA – 1: OC = PEEK(X)
930 PRINT#2,CHR$(OC)940 NEXT X
950 CLOSE2
960 GOTO 820
970 REM ***********
980 REM OPCODE DATA
990 REM ***********
1000 DATA 0,BRK,1
1010 DATA 1,(ORA – X),2
1020 DATA 5,ORA – Z,2
1030 DATA 6,ASL – Z,2
1040 DATA 8,PHP,1
1050 DATA 9,ORA#,2
1060 DATA 10,ASL – A,1
1070 DATA 13,ORA,3
1080 DATA 14,ASL,3
```

```
1090 DATA 16,BPL,2
1100 DATA 17,(ORA – Y),1
1110 DATA 21,ORA – ZX,2
1120 DATA 22,ASL – ZX,2
1130 DATA 24,CLC,1
1140 DATA 25,ORA – Y,3
1150 DATA 29,ORA – X,3
1160 DATA 30,ASL – X,3
1170 DATA 32,JSR,3
1180 DATA 33,(AND – X),2
1190 DATA 36,BIT – Z,2
1200 DATA 37,AND – Z,2
1210 DATA 38,ROL – Z,2
1220 DATA 40,PLP,1
1230 DATA 41,AND#,2
1240 DATA 42,ROL – A,1
1250 DATA 44,BIT,3
1260 DATA 45,AND,3
1270 DATA 46,ROL,3
1280 DATA 48,BMI,2
1290 DATA 49,(AND – Y),2
1300 DATA 53,AND – ZX,2
1310 DATA 54,ROL – ZX,2
1320 DATA 56,SEC,1
1330 DATA 57,AND – Y,3
1340 DATA 61,AND – X,3
1350 DATA 62,ROL – X,3
1360 DATA 64,RTI,1
1370 DATA 65,(EOR – X),2
1380 DATA 69,EOR – Z,2
1390 DATA 70,LSR – Z,2
1400 DATA 72,PHA,1
1410 DATA 73,EOR#,2
1420 DATA 74,LSR – A,1
1430 DATA 76,JMP,3
1440 DATA 77,EOR,3
1450 DATA 78,LSR,3
1460 DATA 80,BVC,2
1470 DATA 81,(EOR – Y),2
1480 DATA 85,EOR – ZX,2
```

```
1490 DATA 86,LSR – ZX,2
1500 DATA 88,CLI,1
1510 DATA 89,EOR – Y,3
1520 DATA 93,EOR – X,3
1530 DATA 94,LSR – X,3
1540 DATA 96,RTS,1
1550 DATA 97,(ADC – X),2
1560 DATA 101,ADC – Z,2
1570 DATA 102,ROR – Z,2
1580 DATA 104,PLA,1
1590 DATA 105,ADC#,2
1600 DATA 106,ROR – A,1
1610 DATA 108,(JMP),3
1620 DATA 109,ADC,3
1630 DATA 110,ROR,3
1640 DATA 112,BVS,2
1650 DATA 113,(ADC – Y),2
1660 DATA 117,ADC – ZX,2
1670 DATA 118,ROR – ZX,2
1680 DATA 120,SEI,1
1690 DATA 121,ADC – Y,3
1700 DATA 125,ADC – X,3
1710 DATA 126,ROR – X,3
1720 DATA 129,(STA – X),2
1730 DATA 132,STY – Z,2
1740 DATA 133,STA – Z,2
1750 DATA 134,STX – Z,2
1760 DATA 136,DEY,1
1770 DATA 138,TXA,1
1780 DATA 140,STY,3
1790 DATA 141,STA,3
1800 DATA 142,STX,3
1810 DATA 144,BCC,2
1820 DATA 145,(STA – Y),2
1830 DATA 148,STY – ZX,2
1840 DATA 149,STA – ZX,2
1850 DATA 150,STX – ZX,2
1860 DATA 152,TYA,1
1870 DATA 153,STA – Y,3
1880 DATA 154,TXS,1
```

```
1890 DATA 157,STA – X,3
1900 DATA 160,LDY#,2
1910 DATA 161,(LDA – X),2
1920 DATA 162,LDX#,2
1930 DATA 164,LDY – Z,2
1940 DATA 165,LDA – Z,2
1950 DATA 166,LDX – Z,2
1960 DATA 168,TAY,1
1970 DATA 169,LDA#,2
1980 DATA 170,TAX,1
1990 DATA 172,LDY,3
2000 DATA 173,LDA,3
2010 DATA 174,LDX,3
2020 DATA 176,BCS,2
2030 DATA 177,(LDA – Y),2
2040 DATA 180,LDY – ZX,2
2050 DATA 181,LDA – ZX,2
2060 DATA 182,LDX – ZY,2
2070 DATA 184,CLV,1
2080 DATA 185,LDA – Y,3
2090 DATA 186,TSX,1
2100 DATA 188,LDY – X,3
2110 DATA 189,LDA – X,3
2120 DATA 190,LDX – Y,3
2130 DATA 192,CPY#,2
2140 DATA 193,(CMP – X),2
2150 DATA 196,CPY – Z,2
2160 DATA 197,CMP – Z,2
2170 DATA 198,DEC – Z,2
2180 DATA 200,INY,1
2190 DATA 201,CMP#,2
2200 DATA 202,DEX,1
2210 DATA 204,CPY,3
2220 DATA 205,CMP,3
2230 DATA 206,DEC,3
2240 DATA 208,BNE,2
2250 DATA 209,(CMP – Y),2
2260 DATA 213,CMP – ZX,2
2270 DATA 214,DEC – ZX,2
2280 DATA 216,CLD,1
```

```
2290 DATA 217,CMP – Y,3
2300 DATA 221,CMP – X,3
2310 DATA 222,DEC – X,3
2320 DATA 224,CPX#,2
2330 DATA 225,(SBC – X),2
2340 DATA 228,CPX – Z,2
2350 DATA 229,SBC – Z,2
2360 DATA 230,INC – Z,2
2370 DATA 232,INX,1
2380 DATA 233,SBC#,2
2390 DATA 234,NOP,1
2400 DATA 236,CPX,3
2410 DATA 237,SBC,3
2420 DATA 238,INC,3
2430 DATA 240,BEQ,2
2440 DATA 241,(SBC – Y),2
2450 DATA 245,SBC – ZX,2
2460 DATA 246,INC – ZX,2
2470 DATA 248,SED,1
2480 DATA 249,SBC – Y,3
2490 DATA 253,SBC – X,3
2500 DATA 254,INC – X,3
2510 REM ******
2520 REM HEADER
2530 REM ******
2540 CR$ = "(C) COPYRIGHT 1984" : NM$ = "BY
WILLIAM B. SANDERS"
2550 BK$ = "ASSEMBLY LANGUAGE FOR KIDS:"
:CM$ = "COMMODORE 64"
2560 IS$ = "SEE" : F$ = "FOR DOCUMENTATION"
2570 H = 20 – LEN(CR$)/2 : PRINT TAB(H);CR$
2580 H = 20 – LEN(NM$)/2 : PRINT TAB(H);NM$
2590 PRINT: H = 20 – LEN(IS$)/2 : PRINT TAB(H);IS$ :
PRINT
2600 H = 20 – LEN(BK$)/2 : PRINT TAB(H);BK$
:H = 20 – LEN(CM$)/2: PRINT TAB(H);CM$
2610 H = 20 – LEN(NM$)/2 : PRINT TAB(H);NM$ :
PRINT
2620 H = 20 – LEN(F$)/2 : PRINT TAB(H);F$
2630 LD$ = "LOADING ARRAY" : FOR X = 1 TO 10 :
```

```
PRINT : NEXT : H = 20 – LEN(LD$)/2
2640 PRINT TAB(H);CHR$(18);LD$
2650 RETURN
```

CASSETTE ENDING ROUTINE

```
740 REM **************
750 REM ENDING ROUTINE
760 REM **************
770 NB = SA – BA
780 PRINT CHR$(147)
790 FOR X = 1 TO 5 : PRINT : NEXT
800 INPUT"SAVE PROGRAM(Y/N)";AN$
810 IF AN$ = "Y" THEN 870
820 PRINT : PRINT : PRINT "PROGRAM
IS";NB;"BYTES LONG"
830 PRINT "TO EXECUTE 'SYS'";BA : PRINT
840 INPUT "(B)EGIN AGAIN OR (E)ND";DE$
850 IF DE$ = "B" THEN 70
860 PRINT : PRINT"END" : END
870 PRINT CHR$(147) : FOR X = 1 TO 5 : PRINT :
NEXT
880 REM *** TAPE SAVE ***
890 INPUT "ENTER FILE NAME";NF$
900 OPEN21,1,1,NF$
910 PRINT#21,BA
920 FOR X = BA TO SA – 1: OC = PEEK(X)
930 PRINT#21,OC
940 NEXT X
950 CLOSE21
960 GOTO 820
```

CASSETTE PROGRAM LOADER

Since you cannot load from tape the same as you can from disk, a special loader program is required. Your assembled program is saved as a SEQ file, and it must be read into memory and then POKEd into memory. The following program will do that for you and show you where it is loaded on the screen:

```
10 PRINT CHR$(147) : X = 0
20 INPUT "NAME OF FILE ";NF$
30 OPEN21,1,0,NF$
40 INPUT#21,BA
50 INPUT#21,OC
60 POKE BA + X,OC
70 PRINT BA + X,OC
80 X = X + 1
90 IF ST = 0 THEN 50
100 CLOSE 21
```

APPENDIX B

6510 OPCODES

Machine Opcodes		Mnemonic Opcodes	Addressing Mode
Dec	**Hex**		
109	$6D	ADC	Absolute
125	$7D	ADC	Absolute,X
121	$79	ADC	Absolute,Y
105	$69	ADC	Immediate
097	$61	ADC	(Indirect,X)
113	$71	ADC	(Indirect),Y
101	$65	ADC	Zero Page
117	$75	ADC	Zero Page,X
045	$2D	AND	Absolute
061	$3D	AND	Absolute,X
057	$39	AND	Absolute,Y
041	$29	AND	Immediate
033	$21	AND	(Indirect,X)
049	$31	AND	(Indirect),Y
037	$25	AND	Zero Page
053	$35	AND	Zero Page,X

Machine Opcodes		Mnemonic Opcodes	Addressing Mode
Dec	Hex		
014	$0E	ASL	Absolute
030	$1E	ASL	Absolute,X
010	$0A	ASL	Accumulator
006	$06	ASL	Zero Page
022	$16	ASL	Zero Page,X
144	$90	BCC	Relative
176	$B0	BCS	Relative
240	$F0	BEQ	Relative
044	$2C	BIT	Absolute
036	$24	BIT	Zero Page
048	$30	BMI	Relative
208	$D0	BNE	Relative
016	$10	BPL	Relative
000	$00	BRK	Implied
080	$50	BVC	Relative
112	$70	BVS	Relative
024	$18	CLC	Implied
216	$D8	CLD	Implied
088	$58	CLI	Implied
184	$B8	CLV	Implied
205	$CD	CMP	Absolute
221	$DD	CMP	Absolute,X
217	$D9	CMP	Absolute,Y
201	$C9	CMP	Immediate
193	$C1	CMP	(Indirect,X)
209	$D1	CMP	(Indirect),Y
197	$C5	CMP	Zero Page
213	$D5	CMP	Zero Page,X
236	$EC	CPX	Absolute
224	$E0	CPX	Immediate
228	$E4	CPX	Zero Page
204	$CC	CPY	Absolute
192	$C0	CPY	Immediate
196	$C4	CPY	Zero Page
206	$CE	DEC	Absolute
222	$DE	DEC	Absolute,X

Machine Opcodes		Mnemonic Opcodes	Addressing Mode
Dec	Hex		
198	$C6	DEC	Zero Page
214	$D6	DEC	Zero Page,X
202	$CA	DEX	Implied
136	$88	DEY	Implied
077	$4D	EOR	Absolute
093	$5D	EOR	Absolute,X
089	$59	EOR	Absolute,Y
073	$49	EOR	Immediate
065	$41	EOR	(Indirect,X)
081	$51	EOR	(Indirect),Y
069	$45	EOR	Zero Page
085	$55	EOR	Zero Page,X
238	$EE	INC	Absolute
254	$FE	INC	Absolute,X
230	$E6	INC	Zero Page
246	$F6	INC	Zero Page
232	$E8	INX	Implied
200	$C8	INY	Implied
076	$4C	JMP	Absolute
108	$6C	JMP	(Indirect)
032	$20	JSR	Absolute
173	$AD	LDA	Absolute
189	$BD	LDA	Absolute,X
185	$B9	LDA	Absolute,Y
169	$A9	LDA	Immediate
161	$A1	LDA	(Indirect,X)
177	$B1	LDA	(Indirect),Y
165	$A5	LDA	Zero Page
181	$B5	LDA	Zero Page,X
174	$AE	LDX	Absolute
190	$BE	LDX	Absolute,Y
162	$A2	LDX	Immediate
166	$A6	LDX	Zero Page
182	$B6	LDX	Zero Page,Y
172	$AC	LDY	Absolute

Machine Opcodes		Mnemonic Opcodes	Addressing Mode
Dec	Hex		
188	$BC	LDY	Absolute,X
160	$A0	LDY	Immediate
164	$A4	LDY	Zero Page
180	$B4	LDY	Zero Page,X
078	$4E	LSR	Absolute
094	$5E	LSR	Absolute,X
074	$4A	LSR	Accumulator
070	$46	LSR	Zero Page
086	$56	LSR	Zero Page,X
234	$EA	NOP	Implied
013	$0D	ORA	Absolute
029	$1D	ORA	Absolute,X
025	$19	ORA	Absolute,Y
009	$09	ORA	Immediate
001	$01	ORA	(Indirect,X)
017	$11	ORA	(Indirect),Y
005	$05	ORA	Zero Page
021	$15	ORA	Zero Page,X
072	$48	PHA	Implied
008	$08	PHP	Implied
104	$68	PLA	Implied
040	$28	PLP	Implied
046	$2E	ROL	Absolute
062	$3E	ROL	Absolute,X
042	$2A	ROL	Accumulator
038	$26	ROL	Zero Page
054	$36	ROL	Zero Page,X
110	$6E	ROR	Absolute
126	$7E	ROR	Absolute,X
106	$6A	ROR	Accumulator
102	$66	ROR	Zero Page
118	$76	ROR	Zero Page,X
064	$40	RTI	Implied
096	$60	RTS	Implied
237	$ED	SBC	Absolute
253	$FD	SBC	Absolute,X

Machine Opcodes		Mnemonic Opcodes	Addressing Mode
Dec	**Hex**		
249	$F9	SBC	Absolute,Y
233	$E9	SBC	Immediate
225	$E1	SBC	(Indirect,X)
241	$F1	SBC	(Indirect),Y
229	$E5	SBC	Zero Page
245	$F5	SBC	Zero Page,X
056	$38	SEC	Implied
248	$F8	SED	Implied
120	$78	SEI	Implied
141	$8D	STA	Absolute
157	$9D	STA	Absolute,X
153	$99	STA	Absolute,Y
129	$81	STA	(Indirect,X)
145	$91	STA	(Indirect),Y
133	$85	STA	Zero Page
149	$95	STA	Zero Page,X
142	$8E	STX	Absolute
134	$86	STX	Zero Page
150	$96	STX	Zero Page,Y
140	$8C	STY	Absolute
132	$84	STY	Zero Page
148	$94	STY	Zero Page,X
170	$AA	TAX	Implied
168	$A8	TAY	Implied
186	$BA	TSX	Implied
138	$8A	TXA	Implied
154	$9A	TXS	Implied
152	$98	TYA	Implied

APPENDIX C

Memory Map 1 : Diagram

$E000-$FFFF 57344-65535	8K Kernal ROM
$D000-$DFFF 53248-57343	4K I/O or Character ROM
$C000-$CFFF 49152-53247	4K RAM
$A000-$BFFF 40960-49151	BASIC ROM or ROM Plug-in
$8000-$9FFF 32768-40959	8K RAM or ROM Plug-in

$4000-$7FFF 16384-32767	16K RAM
$0000-$3FFF 00000-16383	16K RAM
$0800 2048	BASIC begins

APPENDIX D

MEMORY MAP 2 : Places to Visit

This map has been abridged so that you can quickly look up the most-often used subroutines, pointers and free spaces you will be using in assembly language programs. All addresses are given in hexadecimal. Since there are fewer digits in hexadecimal numbers, these values are easier to memorize. (You gotta do it sooner or later.)

Address	What's There
$0-$2A	Misc. Flags and Pointers
$2B-2C	Pointer to beginning of BASIC
$2D-$2E	Pointer to start of variables / end of BASIC program
$2F-$C4	Misc. BASIC flags, functions and pointers
$C5	ASCII of last key pressed
$C6	Number of characters in keyboard buffer
$C7	Screen reverse: 0 = off 18 = on

Address	What's There
$C8-$FA	Misc. flags, functions and pointers
$FB-$FE	Free zero page addresses
$FF-$280	Misc. flags, functions and pointers
$281-$282	Beginning of BASIC memory
$283-$284	Top of memory
$285-$2BF	Misc. flags, functions and pointers
$2C0-$2FF	Block 11 sprite area
$300-$333	Misc. flags, functions and pointers
$334-$33B	Free space
$33C-$3FB	Cassette buffer - MACHINE LANGUAGE STORAGE
$3FC-$3FF	Free space
$340-$37F	Block 13 sprite area
$380-3BF	Block 14 sprite area
$3C0-$3FF	Block 15 sprite area
$400-$7FF	Screen memory 24 x 40
$7F8-$7FF	Sprite pointers for data
$800-$9FFF	BASIC RAM
$8000-$9FFF	Plug in ROM or MACHINE LANGUGE STORAGE
$A000-$BFFF	BASIC ROM
$C000-CFFF	Free RAM - MACHINE LANGUAGE STORAGE
$D000	Sprite 0 X position
$D001	Sprite 0 Y position
$D002	Sprite 1 X position
$D003	Sprite 1 Y position
$D004	Sprite 2 X position
$D005	Sprite 2 Y position
$D006	Sprite 3 X position
$D007	Sprite 3 Y position
$D008	Sprite 4 X position
$D009	Sprite 4 Y position
$D00A	Sprite 5 X position
$D00B	Sprite 5 Y position
$D00C	Sprite 6 X position
$D00D	Sprite 6 Y position
$D00E	Sprite 7 X position

Address	What's There
$D00F	Sprite 7 Y position
$D010	High byte of sprite X position
$D011-$D01F	Misc. flags, functions and pointers
$D020	Border color register
$D021	Background color 0 register
$D022	Background color 1 register
$D023	Background color 2 register
$D024	Background color 3 register
$D025-$D026	Sprite multi-color registers
$D027-$D02E	Sprite color registers
$D400-$D418	Sound registers
$D419	Game paddle 1 or 3
$D41A	Game paddle 2 or 4
$D41B	Random number generator
$D41C-$DD0F	Mis. registers
$E000-$FFFF	Kernal ROM
$E544	Clear screen
$E566	Home cursor in upper left hand corner
$E716	Output to screen
$E8E7	Scroll screen
$FF9F	SCNKEY - scan keyboard
$FFCF	CHRIN - input a character
$FFD2	CHROUT - output a character
$FFE4	GETIN - get a character
$FFF0	PLOT - read or set X,Y postion of cursor

APPENDIX E

BASIC TOKEN CHART

Your Commodore 64 reads BASIC statements as tokenized codes. In a disassembled listing of a BASIC program, the following tokenized values can be found representing the BASIC statements.

DEC HEX Keybd.	DEC HEX Keybd.
128 $80 - END	165 $A5 - FN
129 $81 - FOR	166 $A6 - SPC(
130 $82 - NEXT	167 $A7 - THEN
131 $83 - DATA	168 $A8 - NOT
132 $84 - INPUT#	169 $A9 - STEP
133 $85 - INPUT	170 $AA - +
134 $86 - DIM	171 $AB - -
135 $87 - READ	172 $AC - *
136 $88 - LET	173 $AD - /
137 $89 - GOTO	174 $AE -
138 $8A - RUN	175 $AF - AND
139 $8B - IF	176 $B0 - OR
140 $8C - RESTORE	177 $B1 -
141 $8D - GOSUB	178 $B2 - =

DEC HEX Keybd.	DEC HEX Keybd.
142 $8E - RETURN	179 $B3 -
143 $8F - REM	180 $B4 - SGN
144 $90 - STOP	181 $B5 - INT
145 $91 - ON	182 $B6 - ABS
146 $92 - WAIT	183 $B7 - USR
147 $93 - LOAD	184 $B8 - FRE
148 $94 - SAVE	185 $B9 - POS
149 $95 - VERIFY	186 $BA - SQR
150 $96 - DEF	187 $BB - RND
151 $97 - POKE	188 $BC - LOG
152 $98 - PRINT#	189 $BD - EXP
153 $99 - PRINT	190 $BE - COS
154 $9A - CONT	191 $BF - SIN
155 $9B - LIST	192 $C0 - TAN
156 $9C - CLR	193 $C1 - ATN
157 $9D - CMD	194 $C2 - PEEK
158 $9E - SYS	195 $C3 - LEN
159 $9F - OPEN	196 $C4 - STR$
160 $A0 - CLOSE	197 $C5 - VAL
161 $A1 - GET	198 $C6 - ASC
162 $A2 - NEW	199 $C7 - CHR$
163 $A3 - TAB(	200 $C8 - LEFT$
164 $A4 - TO	201 $C9 - RIGHT$
	202 $CA - MID$

APPENDIX F

HEXADECIMAL-DECIMAL CONVERSION

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
$00 =	0	$33 =	51	$66 =	102	$99 =	153	$CC =	204
$01 =	1	$34 =	52	$67 =	103	$9A =	154	$CD =	205
$02 =	2	$35 =	53	$68 =	104	$9B =	155	$CE =	206
$03 =	3	$36 =	54	$69 =	105	$9C =	156	$CF =	207
$04 =	4	$37 =	55	$6A =	106	$9D =	157	$D0 =	208
$05 =	5	$38 =	56	$6B =	107	$9E =	158	$D1 =	209
$06 =	6	$39 =	57	$6C =	108	$9F =	159	$D2 =	210
$07 =	7	$3A =	58	$6D =	109	$A0 =	160	$D3 =	211
$08 =	8	$3B =	59	$6E =	110	$A1 =	161	$D4 =	212
$09 =	9	$3C =	60	$6F =	111	$A2 =	162	$D5 =	213
$0A =	10	$3D =	61	$70 =	112	$A3 =	163	$D6 =	214
$0B =	11	$3E =	62	$71 =	113	$A4 =	164	$D7 =	215
$0C =	12	$3F =	63	$72 =	114	$A5 =	165	$D8 =	216
$0D =	13	$40 =	64	$73 =	115	$A6 =	166	$D9 =	217
$0E =	14	$41 =	65	$74 =	116	$A7 =	167	$DA =	218
$0F =	15	$42 =	66	$75 =	117	$A8 =	168	$DB =	219
$10 =	16	$43 =	67	$76 =	118	$A9 =	169	$DC =	220
$11 =	17	$44 =	68	$77 =	119	$AA =	170	$DD =	221

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
$12 =	18	$45 =	69	$78 =	120	$AB =	171	$DE =	222
$13 =	19	$46 =	70	$79 =	121	$AC =	172	$DF =	223
$14 =	20	$47 =	71	$7A =	122	$AD =	173	$E0 =	224
$15 =	21	$48 =	72	$7B =	123	$AE =	174	$E1 =	225
$16 =	22	$49 =	73	$7C =	124	$AF =	175	$E2 =	226
$17 =	23	$4A =	74	$7D =	125	$B0 =	176	$E3 =	227
$18 =	24	$4B =	75	$7E =	126	$B1 =	177	$E4 =	228
$19 =	25	$4C =	76	$7F =	127	$B2 =	178	$E5 =	229
$1A =	26	$4D =	77	$80 =	128	$B3 =	179	$E6 =	230
$1B =	27	$4E =	78	$81 =	129	$B4 =	180	$E7 =	231
$1C =	28	$4F =	79	$82 =	130	$B5 =	181	$E8 =	232
$1D =	29	$50 =	80	$83 =	131	$B6 =	182	$E9 =	233
$1E =	30	$51 =	81	$84 =	132	$B7 =	183	$EA =	234
$1F =	31	$52 =	82	$85 =	133	$B8 =	184	$EB =	235
$20 =	32	$53 =	83	$86 =	134	$B9 =	185	$EC =	236
$21 =	33	$54 =	84	$87 =	135	$BA =	186	$ED =	237
$22 =	34	$55 =	85	$88 =	136	$BB =	187	$EE =	238
$23 =	35	$56 =	86	$89 =	137	$BC =	188	$EF =	239
$24 =	36	$57 =	87	$8A =	138	$BD =	189	$F0 =	240
$25 =	37	$58 =	88	$8B =	139	$BE =	190	$F1 =	241
$26 =	38	$59 =	89	$8C =	140	$BF =	191	$F2 =	242
$27 =	39	$5A =	90	$8D =	141	$C0 =	192	$F3 =	243
$28 =	40	$5B =	91	$8E =	142	$C1 =	193	$F4 =	244
$29 =	41	$5C =	92	$8F =	143	$C2 =	194	$F5 =	245
$2A =	42	$5D =	93	$90 =	144	$C3 =	195	$F6 =	246
$2B =	43	$5E =	94	$91 =	145	$C4 =	196	$F7 =	247
$2C =	44	$5F =	95	$92 =	146	$C5 =	197	$F8 =	248
$2D =	45	$60 =	96	$93 =	147	$C6 =	198	$F9 =	249
$2E =	46	$61 =	97	$94 =	148	$C7 =	199	$FA =	250
$2F =	47	$62 =	98	$95 =	149	$C8 =	200	$FB =	251
$30 =	48	$63 =	99	$96 =	150	$C9 =	201	$FC =	252
$31 =	49	$64 =	100	$97 =	151	$CA =	202	$FD =	253
$32 =	50	$65 =	101	$98 =	152	$CB =	203	$FE =	254
$33 =	51	$66 =	102	$99 =	153	$CC =	204	$FF =	255

APPENDIX G

DECIMAL-HEXADECIMAL CONVERSION CHART

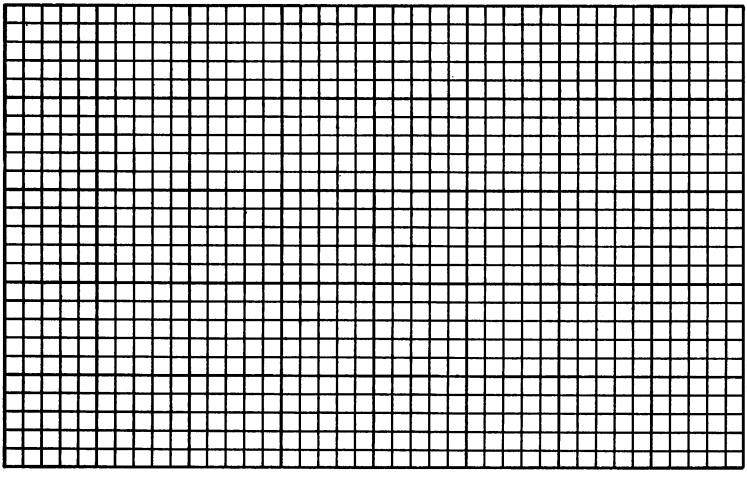
DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX
0	= $00	51	= $33	102	= $66	153	= $99	204	= $CC
1	= $01	52	= $34	103	= $67	154	= $9A	205	= $CD
2	= $02	53	= $35	104	= $68	155	= $9B	206	= $CE
3	= $03	54	= $36	105	= $69	156	= $9C	207	= $CF
4	= $04	55	= $37	106	= $6A	157	= $9D	208	= $D0
5	= $05	56	= $38	107	= $6B	158	= $9E	209	= $D1
6	= $06	57	= $39	108	= $6C	159	= $9F	210	= $D2
7	= $07	58	= $3A	109	= $6D	160	= $A0	211	= $D3
8	= $08	59	= $3B	110	= $6E	161	= $A1	212	= $D4
9	= $09	60	= $3C	111	= $6F	162	= $A2	213	= $D5
10	= $0A	61	= $3D	112	= $70	163	= $A3	214	= $D6
11	= $0B	62	= $3E	113	= $71	164	= $A4	215	= $D7
12	= $0C	63	= $3F	114	= $72	165	= $A5	216	= $D8
13	= $0D	64	= $40	115	= $73	166	= $A6	217	= $D9
14	= $0E	65	= $41	116	= $74	167	= $A7	218	= $DA
15	= $0F	66	= $42	117	= $75	168	= $A8	219	= $DB
16	= $10	67	= $43	118	= $76	169	= $A9	220	= $DC
17	= $11	68	= $44	119	= $77	170	= $AA	221	= $DD

DEC HEX	DEC HEX	DEC HEX	DEC HEX	DEC HEX
18 = $12	69 = $45	120 = $78	171 = $AB	222 = $DE
19 = $13	70 = $46	121 = $79	172 = $AC	223 = $DF
20 = $14	71 = $47	122 = $7A	173 = $AD	224 = $E0
21 = $15	72 = $48	123 = $7B	174 = $AE	225 = $E1
22 = $16	73 = $49	124 = $7C	175 = $AF	226 = $E2
23 = $17	74 = $4A	125 = $7D	176 = $B0	227 = $E3
24 = $18	75 = $4B	126 = $7E	177 = $B1	228 = $E4
25 = $19	76 = $4C	127 = $7F	178 = $B2	229 = $E5
26 = $1A	77 = $4D	128 = $80	179 = $B3	230 = $E6
27 = $1B	78 = $4E	129 = $81	180 = $B4	231 = $E7
28 = $1C	79 = $4F	130 = $82	181 = $B5	232 = $E8
29 = $1D	80 = $50	131 = $83	182 = $B6	233 = $E9
30 = $1E	81 = $51	132 = $84	183 = $B7	234 = $EA
31 = $1F	82 = $52	133 = $85	184 = $B8	235 = $EB
32 = $20	83 = $53	134 = $86	185 = $B9	236 = $EC
33 = $21	84 = $54	135 = $87	186 = $BA	237 = $ED
34 = $22	85 = $55	136 = $88	187 = $BB	238 = $EE
35 = $23	86 = $56	137 = $89	188 = $BC	239 = $EF
36 = $24	87 = $57	138 = $8A	189 = $BD	240 = $F0
37 = $25	88 = $58	139 = $8B	190 = $BE	241 = $F1
38 = $26	89 = $59	140 = $8C	191 = $BF	242 = $F2
39 = $27	90 = $5A	141 = $8D	192 = $C0	243 = $F3
40 = $28	91 = $5B	142 = $8E	193 = $C1	244 = $F4
41 = $29	92 = $5C	143 = $8F	194 = $C2	245 = $F5
42 = $2A	93 = $5D	144 = $90	195 = $C3	246 = $F6
43 = $2B	94 = $5E	145 = $91	196 = $C4	247 = $F7
44 = $2C	95 = $5F	146 = $92	197 = $C5	248 = $F8
45 = $2D	96 = $60	147 = $93	198 = $C6	249 = $F9
46 = $2E	97 = $61	148 = $94	199 = $C7	250 = $FA
47 = $2F	98 = $62	149 = $95	200 = $C8	251 = $FB
48 = $30	99 = $63	150 = $96	201 = $C9	252 = $FC
49 = $31	100 = $64	151 = $97	202 = $CA	253 = $FD
50 = $32	101 = $65	152 = $98	203 = $CB	254 = $FE
51 = $33	102 = $66	153 = $99	204 = $CC	255 = $FF

APPENDIX H

SCREEN STORAGE ADDRESS TABLE

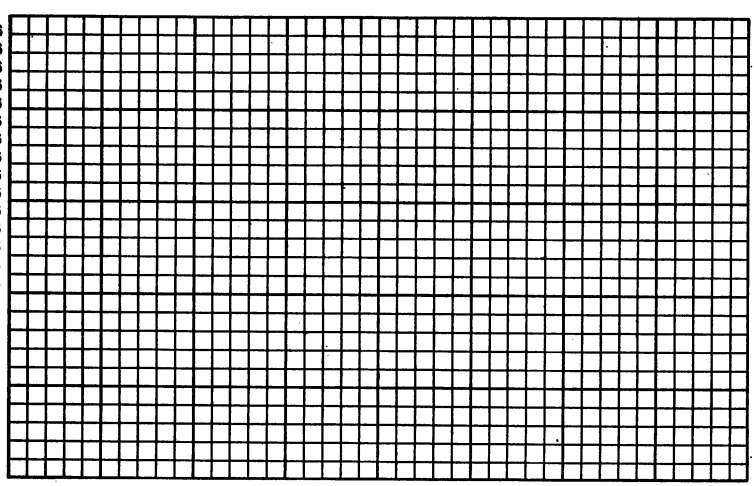
$400	1024
$428	1064
$450	1104
$478	1144
$4A0	1184
$4C8	1224
$4F0	1264
$518	1304
$540	1344
$568	1384
$590	1424
$5B8	1464
$5E0	1504
$608	1544
$630	1584
$658	1624
$680	1664
$6A8	1704
$6D0	1744
$6F8	1784
$720	1824
$748	1864
$770	1904
$798	1944
$7C0	1984



APPENDIX I

COLOR STORAGE LOCATION TABLE

$D800	55296
$D828	55336
$D878	55416
$D8A0	55456
$D8C8	55496
$D8F0	55536
$D918	55576
$D940	55616
$D968	55656
$D990	55696
$D9B8	55736
$D9E0	55776
$DA08	55816
$DA30	55856
$DA58	55896
$DA80	55936
$DAA8	55976
$DAD0	56016
$DAF8	56056
$DB20	56096
$DB48	56136
$DB70	56176
$DB98	56216
$DBC0	56256
$DBE8	56296



APPENDIX J

ASCII CODE

These characters appear on the screen when sent from the accumulator using the CHROUT subroutine or output to screen routines (JSR $E716). See APPENDIX K for screen codes that appear when the code is output with STA to a screen address (1024-2023).

All values to the left are the ASCII values, and all characters, symbols and descriptions to the right are screen displays

0	51 3	102	153 Lt Green	204	
1	52 4	103	154 Lt Blue	205	
2	53 5	104	155 Gray 3	206	
3	54 6	105	156 Purple	207	
4	55 7	106	157 CRSR left	208	
5 White	56 8	107	158 Yellow	209	
6	57 9	108	159 Cyan	210	
7	58 :	109	160 SPACE	211	♥
8 Sh-CMD off	59 ;	110	161	212	
9 Sh-CMD on	60 <	111	162	213	
10	61 =	112	163	214	
11	62 >	113	164	215	

12
13 RETURN
14 Lowercase
15
6
17 CRSR down
18 RVS on
19 Home CRSR
20 Delete
21
22
23
24
25
26
27
28 Red
29 CRSR right
30 Green
31 Blue
32 SPACE
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
49 1
50 2

63 ?
64 @
65 A
66 B
67 C
68 D
69 E
70 F
71 G
72 H
73 I
74 J
75 K
76 L
77 M
78 N
79 O
80 P
81 Q
82 R
83 S
84 T
85 U
86 V
87 W
88 X
89 Y
90 Z
91 [
92 £
93]
94 ↑
95 ←
96
97
98
99
100
101

114
115
116
117
118
119
120
121
122
123
124
125
126 π
127
128
129 Orange
130
131
132
133 f1
134 f3
135 f5
136 f7
137 f2
138 f4
139 f6
140 f8
141 Sh-RETURN
142 Uppercase
143
144 Black
145 CRSR up
146 RVS off
147 CLR/HOME
148 INST
149 Brown
150 Lt Red
151 Gray 1
152 Gray 2

165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203

216
217
218
219
220
221
222 π
223
224 SPACE
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255 π

APPENDIX K

SCREEN STORAGE DISPLAY CODES

When values are stored in addresses 1024-2023 ($400-$7E7) they will appear as the characters in this chart. The first set is upper case and full graphics (UC), and the second set (UC/LC) is upper case and lower case. If your keyboard is set to upper case and full graphics, you will get the characters in the first set, and if it is set to upper/lower case, you will get the second. Thus, any value that you STA (or STX or STY) in these addresses will show up on the screen as alphanumeric or graphic characters

#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC
0	@	@	51	3	3	102			153			204		
1	A	a	52	4	4	103			154			205		
2	B	b	53	5	5	104			155			206		
3	C	c	54	6	6	105			156			207		
4	D	d	55	7	7	106			157			208		
5	E	e	56	8	8	107			158			209		
6	F	f	57	9	9	108			159			210		
7	G	g	58	:	:	109			160			211		
8	H	h	59	;	;	110			161			212		
9	I	i	60	<	<	111			162			213		
10	J	j	61	=	=	112			163			214		

#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC	#	UC	UC/LC
11	K	k	62	►	►	113		
12	L	l	63	?	?	114		
13	M	m	64			115		
14	N	n	65	♠	A	116		
15	O	o	66		B	117		
16	P	p	67		C	118		
17	Q	q	68		D	119		
18	R	r	69		E	120		
19	S	s	70		F	121		
20	T	t	71		G	122		
21	U	u	72		H	123		
22	V	v	73		I	124		
23	W	w	74		J	125		
24	X	x	75		K	126		
25	Y	y	76		L	127		
26	Z	z	77		M	128 - 255 reverse video of 0-127		
27			78		N			
28			79		O			
29			80		P			
30			81		Q			
31			82		R			
32	SPACE		83	♥	S			
33	!	!	84		T			
34	"	"	85		U			
35	#	#	86		V			
36	$	$	87		W			
37	%	%	88	♣	X			
38	&	&	89		Y			
39	'	'	90	♦	Z			
40	(	(	91					
41	)	)	92					
42	*	*	93					
43	+	+	94	π				
44	,	,	95					
45	-	-	96					
46	.	.	97					
47	/	/	98					
48	0	0	99					
49	1	1	100					
50	2	2	101					

INDEX

This INDEX covers the major references to various key terms. Many of the opcodes were used throughout the book in several programs, but their only reference in the Index are to those places where pertinent information was introduced or elaborated. Terms in the Appendices are not in the Index but are located at the beginning of the Appendices.

A-B

ABSOLUTE MODE 132-133, 184
ACCUMULATOR 104-105
ADC 186-189
ADDRESSING MODES 123
ANIMATION 230-237
APPEND 151-153, 280-290
ASC 211-215
ASCII 200, 237
ASSEMBLE SOURCE CODE 56
ASSEMBLERS 5, 21-26, 24
ASSEMBLER64 80-82
ASSEMBLY LANGUAGE 6-8
BASIC 10-15
BASIC LOGIC 167
BEQ 170, 177-181
BINARY NUMBERS 91-101
BNE 170, 177-181
BOOKS 291-293
BRANCH 166-167, 171, 177-181, 179
BREAK FLAG 107
.BYTE (directive) 211-215, 262, 269
BYTE 123-124

C-D

CARRY FLAG 108
CHANGE 63, 79, 285
CHROUT 127, 195-197, 199, 225-227, 238
CLC 187-189,226
CMP 169

COLOR 200-201, 222
COLOR CODES 19
COMMODORE 64 MACRO ASSEMBLER 73-89
COPY 64-65
CPX 169
CPY 169
DEC 184
DECIMAL FLAG 107
DELETE 61 80
DEX 148-150
DEY 148-150
DFB 262, 264-265, 269
DISASSEMBLER 26
DOS WEDGE64 73,78

E-F
EDITORS 23-24
EDITOR64 73-80, 280
EOR 205-207
FIELDS 27-28
FIND 80
FLAGS 105-108
FORMAT 76, 89

G-H
GET 79
GETIN 127, 194-195
GRAPHICS 219-246
GROTKE-GUY 37, 138
HEADER 77
HEXADECIMAL NUMBERS 91-101
HIGH BYTE 35, 97, 121-123
HIGH NIBBLE 99

I-J
IMMEDIATE MODE 132-133
IMPLIED MODE 132-133
INC 184-186
INDEXED ABSOLUTE MODE 154-156
INDEXED INDIRECT MODE 157-160
INDEXED MODE 172-177, 185
INDIRECT INDEXED MODE 161-163
INSERT 60, 285
INTERRUPT FLAG 107
INX 148-150
INY 148-150
JMP 177-181
JOYSTICK 193, 203-211, 237-246, 285
JSR 126-128, 198-199

K-L
KERNAL 127
KEYBOARD 194-203
KIDS' ASSEMBLER 29-51
LABEL FIELD 27
LABELS 195
LDA 129-132
LINE NUMBERS 113-116
LIST 59
LOAD PROGRAM 36-38, 67, 79, 138-139
LOADERS 82-83
LOADERS AND SAVERS 25
LOADING PROGRAMS 48-50
LOOPS 166, 168-177, 178
LOW BYTE 35, 97, 121-123
LOW NIBBLE 99
LOW RESOLUTION GRAPHICS 220-228

M-N
MACHINE LANGUAGE 6-8
MAGAZINES 293-294
MEMORY 116-119, 126, 134-142
MERGING SUBROUTINES 280-290
MERLIN64 ADD MODE 56
MERLIN64 COMMAND MODE 55
MERLIN64 EDITOR 55, 59-66
MERLIN64 EXECUTIVE MODE 54
MERLIN64 53-72
MESSAGE MAKER 215-217
MNEMONIC 5
MONITOR 25-26, 70, 83-88, 119-121
MOVE 65, 285
NEGATIVE FLAG 106
NESTED LOOPS 175-177
NUMBER CONVERSION 91-101

O-P
OBJECT CODE 24

OPCODES 5, 22, 23, 30-31, 39-41, 47-48, 123-124, 289-290
OPCODE FIELD 44
OPERANDS 22, 23, 32
OPERAND FIELD 44
ORG 125-126
OVERFLOW FLAG 107
PEEK 11-12, 219, 247
PLOT 127, 225-229, 232, 237
POKE 16-17, 219, 247
PRG FILE 38, 37
PRINTER 62-63
PROCESSOR STATUS REGISTER 105-106
PROGRAM COUNTER 110
PUT 78

R-S
RAM 44
REGISTERS 103-111
RENUMBER 76-77
REPLACE 65-66
SAVE CODE 78
SAVE FILE 57, 86
SAVE GRAPHICS 228-229
SAVE OBJECT CODE 58-59
SAVE SOURCE CODE 57
SBC 186, 190-191
SCNKEY 127, 194-195
SCREEN ADDRESSES 140-141
SEC 187, 190-191
SEQ FILE 36
SEQUENTIAL STRUCTURE 165
SOFT SWITCHES 135-137
SOUND 274-278
SOUND REGISTERS 275
SOURCE CODE 23
SOURCEROR 68-69
SPRITE & SOUND 84, 281-282 84
SPRITE ASSEMBLER 265-267
SPRITE COLOR 254
SPRITE CREATION 251-256
SPRITE ENABLE 253-254
SPRITE EXPANSION 272-274
SPRITE MATRIX 248
SPRITE MOVEMENT 255-256, 272
SPRITE POINTERS 253
SPRITE STORAGE 252
SPRITES 247-274
STA 134, 150
STACK POINTER 108-110
STRUCTURE 165-181
STX 150
STY 150
SUBROUTINES 13-15, 126-128
SYS 25, 37, 46, 59, 193, 228

T-U
TAY 146
TOKENS 12
TXA 146-148
TYA 146-148
TYPES OF ASSEMBLERS 8-9
USER GROUPS 290-291

X-Z
X REGISTER 105, 143-163, 172, 223
Y REGISTER 105, 143-163, 172, 222
ZERO FLAG 107
ZERO PAGE 160

Like printing your own money...
SPECIAL 10% REBATE OFFER

Now that you have taken the first step toward learning how to program in assembly language, why not **do yourself a favor** and get **Merlin 64,** the best assembler available for the Commodore 64, and **save a little money** at the same time!

Merlin 64 features includes nestable **macros,** assemble to disk, use of linked source files, over 35 psuedo opcodes, handy crossreference utilities, a powerful **Editor,** and a **Monitor** to move, compare, disassemble and dump blocks of memory. Merlin 64 also includes **Sourceror,** an easy to use disassembler that creates Merlin 64 source files for editing from binary programs.

And here's the best part. We are making you **"an offer you can't refuse!"** That's right! We are offering to **pay you to use** Merlin 64. Merlin 64 will make your assembly language programming a breeze, and put some cash back into your pockets in the process.

And, just like Merlin 64, the rebate is easy to use. Just fill out and send in this certificate to receive a **"ONE TIME ONLY" 10%** (ten percent) **REBATE** of the net purchase price (excluding state or local taxes) of Merlin 64.

Just send us your **original, itemized sales receipt** or invoice (as proof of purchase) and the completed **Software Registration Form** enclosed in each package accompanied by this certificate and **we'll write you a check!**

Your receipt will be returned to you along with your rebate check. For your protection, we recommend that you make a photocopy of the sales receipt prior to mailing.

(Please print)
NAME: _____

ADDRESS: _____

CITY, STATE, ZIP: _____

PHONE: ( ) _____

Net Purchase Price $ _____ Rebate Amount Due $ _____

(Your Signature) _____

OUR GUARANTEE

Roger Wagner Publishing products carry the unconditional guarantee of satisfaction or your money back. Any product may be returned to place of purchase for complete refund or replacement within thirty (30) days of purchase if accompanied by the sales receipt or other proof of purchase.

Roger Wagner Publishing
P.O. Box 582, 10761-E Woodside Ave.
Santee, CA 92071
PH: (619) 562-3670

Rebate offer and prices as of 8/1/84, subject to change without notice.

= = KIDS' ASSEMBLER ON DISK = =

ONLY $1Ø

If you don't feel like keying in the Kids' Assembler, you can get it cheap from MICROCOMSCRIBE. There's nothing new on the disk that's not in the book, except it's an inexpensive way to save time keying in the assemblers, supporting programs and avoiding typos you might make. The compiled version of the Kids' Assembler runs a lot faster than the BASIC version.

Here's what you get:

KIDS ASSEMBLER1
KIDS ASSEMBLER2
KIDS ASSEMBLER1-C (Cassette)
KIDS ASSEMBLER2-M (Compiled-Disk only)
SOURCE READER-D (Disk)
SOURCE READER-C (Cassette)
CASSETTE LOADER (Cassette)
HEX-DEC CONVERTER
BINARY-DEC CONVERTER

Fill out the following coupon:

NAME _____

ADDRESS _____

CITY _____

STATE _____ ZIP _____

Send coupon and $1Ø to:

MICROCOMSCRIBE
8982 STIMSON COURT
SAN DIEGO, CA 92129

(California residents add 6% sales tax. $1Ø.6Ø total.)

ONLY disks will be mailed. If you have a cassette system, be sure you can borrow a disk drive to download the programs to your tape. (Better yet, save the ten bucks toward a purchase of a disk drive.)

USER GROUPS and EDUCATIONAL INSTITUTIONS will receive special considerations. Write MICROCOMSCRIBE for details.

%%%%%% OFFER EXPIRES 12/31/86 %%%%%%%%

ASSEMBLY LANGUAGE
FOR KIDS
COMMODORE 64

by
WILLIAM B. SANDERS

LEARN ASSEMBLY LANGUAGE PROGRAMMING If you'd rather be one of the kids who writes professional quality arcade games than one who just plays them, learn machine/assembly language programming.

WHAT COMPUTER? Everything in this book is for the **Commodore 64**. You'll get the right information for *your* computer; not everyone else's.

WHO'S THIS BOOK FOR? If you know BASIC and want to learn the fastest language in programming; this book is for you. (If you're an adult, tell them you got it for your nephew in Borneo.) This is an *elementary* book for learning to use assemblers and assembly/machine language programming on your **Commodore 64.**

WHICH ASSEMBLER? Three assemblers are fully covered, and most others for the **Commodore 64** are compatible with all programs. Commodore's assembler, *The Commodore 64 Macro Assembler Development System*, is *clearly* explained with lots of examples. The *Merlin* assembler is *clearly* explained with lots of examples. If you don't have an assembler, there's a simple-to-learn and use listing of the *Kids' Assembler* written in BASIC for you *free* in the book. The *Kids' Assembler* assembles programs for you, *and* it will help you learn about assembly/machine language programming.

WHAT YOU GET

- An Assembler and instructions on using the most popular **Commodore 64** assemblers.
- Charts covering everything from hexadecimal – decimal conversions to BASIC tokens to 6510 opcodes.
- Step-by-step, clear explanations and clear examples of assembly language programming.
- Practical utility programs in assembly/machine language you can write yourself (and understand!).
- Amazing graphics, stupendous sprites, booming sounds, blinding speed, fame, fortune and a heck of a lot of fun.

Written by, William B. Sanders, the author of the best-selling *Elementary Commodore 64;* you will learn assembly language more simply than you thought possible.

microcomscribe
Literate Microcomputer Documentation

ISBN 0-931145-00-7