Radio Shack®

Cat. No. 62-2077

TRS-80® Color Computer Assembly Language Programming

By William Barden, Jr.

■ A comprehensive guide to 6809 instruction set and registers ■ Storing data to memory ■ Condition codes ■ Addressing and branching ■ Subroutines ■ Arithmetic operations ■ ROM subroutines ■ Graphics and sound

COLOR COMPUTER ASSEMBLY LANGUAGE PROGRAMMING

by

William Barden, Jr.

FIRST EDITION PRINTING 9 8 7 6 5 4 3 2

International Standard Book Number: Library of Congress Book Number:

Printed in USA

Preface

The Color Computer is an exceptionally good machine on which to learn assembly language.

For one thing, it uses a 6809 microprocessor. The 6809 microprocessor is a relatively new microprocessor with a powerful instruction set and addressing modes. The 6809 has many features that aren't found in older microprocessors such as the Z-80, 6800, and 6502 — completely "position independent" code, which can be moved anywhere in memory without changing values, powerful "indexing" capability with "auto increment and decrement" to automatically adjust the index registers by one or two counts, a "user" stack pointer defining a user stack area, and a built-in Multiply instruction.

Another feature that makes assembly-language programming attractive on the Color Computer is the availability of EDTASM+, the Radio Shack Editor/Assembler/Debug package. This is a ROM-pack assembler that allows the three functions to be resident in memory at one time. This means that you can switch from Edit mode to Assembly mode in the wink of an eye, and then immediately switch to the Debug portion of EDTASM+ (ZBUG) for program checking. After debugging, you can get back to the Editor and Assembler just as easily.

The interactiveness of EDTASM+ makes it very easy to learn assembly-language programming, as there are no long time delays loading three different programs to Edit, Assemble, and Debug.

Finally there's the Color Computer itself. I feel that it is highly underrated. Much more than a "game machine," it can do some powerful programming. And, of course, it has a built-in high-resolution color graphics capability, sound generation capability, and other input/output ports, all of which can be accessed with assembly language in addition to BASIC.

In this book we've tried to make it easy for you to learn 6809/Color Computer Assembly Language. We'll take you from the ground up (although some knowledge of BASIC is a definite help). There're plenty of interactive examples here, and we've made wide use of EDTASM+ capabilities.

Color Computer Assembly Language Programming has 26 chapters.

The first 18 chapters cover assembly-language basics, the instruction set of the 6809, the basic addressing modes of the 6809, and use of EDTASM+.

The next 3 chapters describe how to interface assembly-language programs to BASIC — how to link the two types of code, how to pass "parameters," and where in memory to put assembly-language code.

Chapter 22 discusses BASIC ROM subroutines, segments of predefined code in the BASIC interpreter that can be used by the assembly-language programmer to do things such as displaying characters on the screen and reading keyboard data.

Chapter 23 describes the more advanced addressing modes of the 6809. While these modes do not have to be used in assembly-language programs, that do offer powerful "addressing" options that are not found in many microprocessors.

The next two chapters, 24 and 25, are the "fun" chapters. These cover video and graphics work and sound generation. Included are a music synthesizer program with variable "envelope" and a "zoom" capability for graphics.

The last chapter, 26, gives some guidelines on writing large assembly-language programs. It covers the procedures of design, flowcharting, coding, debugging, and documentation.

Six appendices are also included in the book. Appendix I provides an alphabetized list of instructions with a capsulized description of each. Appendix II gives a "crib-sheet" type listing of 6809 instructions in more detail. Appendix III is a complete listing of EDTASM+ commands. Appendix IV is a conversion chart of hexadecimal, binary, and decimal numbers from 0 through 255 decimal. Appendix V is a listing of "two's complement" numbers from +127 through -128 decimal. Finally, Appendix VI is a list of ASCII codes for the Color Computer.

Each chapter is preceded by a "Key Chart," which lists the material covered in the chapter, plus the material covered in preceding chapters. The Key Chart helps in finding a particular topic quickly.

I've had a lot of fun writing this book and hope that you enjoy Color Computer Assembly-Language Programming as much I have. See you at $3F00!

William Barden, Jr.

To Aunts Katherine Wilke and Jeanne Damp and Uncles Larry Wilke and Kelly Damp. (Letters follow if I have my relative addressing correct.)

Table of Contents

Appendices

KEY CHART — CHAPTER 1

INSTRUCTIONS

ABX	LBHS	CLRB	LDS	ROL
ADCA	BITA	CLR	LDU	RORA
ADCB	BITB	CMPA	LDX	RORB
ADDA	BLE	CMPB	LDY	ROR
ADDB	LBLE	CMPD	LEAS	RTI
ADDD	BLO	CMPS	LEAU	RTS
ANDA	LBLO	CMPU	LEAX	SBCA
ANDB	BLS	CMPX	LEAY	SBCB
ANDCC	LBLS	CMPY	LSLA	SEX
ASLA	BLT	COMA	LSLB	STA
ASLB	LBLT	COMB	LSL	STB
ASL	BMI	COM	LSRA	STD
ASRA	LBMI	CWAI	LSRB	STS
ASRB	BNE	DAA	LSR	STU
ASR	LBNE	DECA	MUL	STX
BCC	BPL	DECB	NEGA	STY
LBCC	LBPL	DEC	NEGB	SUBA
BCS	BRA	EORA	NEG	SUBB
LBCS	LBRA	EORB	NOP	SUBD
BEQ	BRN	EXG	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	JMP	PSHU	TFR
LBGT	LBVC	JSR	PULS	TSTA
BHI	BVS	LDA	PULU	TSTB
LBHI	LBVS	LDB	ROLA	TST
BHS	CLRA	LDD	ROLB	

ADDRESSING MODES

INHERENT
DIRECT
EXTENDED
IMMEDIATE
SIMPLE INDEXED
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	SETDP

EDTASM+ EDITOR COMMANDS

A(SSEMBLE)	**I(NSERT)**	R(EPLACE)
C(OPY)	**L(OAD)**	**T(HARDCOPY)**
D(ELETE)	M(OVE)	**V(ERIFY)**
E(DIT)	**N(UMBER)**	**W(RITE)**
F(IND)	**P(RINT)**	**Z(BUG)**
H(ARDCOPY)	**Q(UIT)**	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
/IM IN MEMORY ASSEMBLY	/NS NO SYMBOL TABLE
/LP LINE PRINTER	/SS SHORT SCREEN
/MO MANUAL ORIGIN	**/WE WAIT ON ERRORS**
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	T DISPLAY BLOCK
B(YTE) MODE	T H HARDCOPY BLOCK
C(ONTINUE)	U MOVE BLOCK
D(ISPLAY)	V (ERIFY) BLOCK
E(DITOR)	W(ORD) MODE
G(O)	**X BREAKPOINT**
H(ALF) SYMBOLIC	Y (ANK) BREAKPOINT
I(NPUT) BASE	↑ EXAMINE PRECEDING
L(OAD) ML FILE	↓ EXAMINE NEXT
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
O(OUTPUT) BASE	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
R(EGISTER) DISPLAY	/ EXAMINE
S(YMBOLIC DISPLAY)	, SINGLE STEP

GENERAL TOPICS

CPU REGISTERS	SUBROUTINES
DATA TO REGISTERS	STACK OPERATIONS
LOADING AND STORING	ROTATES, SHIFTS
ADDITION AND SUBTRACTION	MULTIPLES
CONDITION CODES	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ *= Past Chapters*

Chapter 1
6809/Color Computer Assembly Language

To start off, we'll look at what 6809 assembly language is and how EDTASM+ converts an assembly-language program to machine-language instructions. We'll also look at some of the Edit Mode commands in EDTASM+ that allow us to enter assembly-language programs.

The 6809 Instruction Set

The 6809 microprocessor used in the Color Computer has a built-in "instruction set." This instruction set consists of dozens of different types of instructions.

Hints and Kinks 1-1
Why Do We Say 6809 Instead of 6809E?

The 6809E is a special version of the 6809 microprocessor. The "E" stands for external clock. We'll just use the generic name "6809" whenever we're talking about the microprocessor in the Color Computer, even though it should be strictly called the "E."

Computers started out as fast adding machines, and for that reason, a lot of the instructions are oriented toward arithmetic operations. Instructions that add two numbers or subtract two numbers are common.

There are other types of instructions that are related to program flow — such instructions as "Branch to a Location" and "Branch if a Zero Result."

Part of the problem in learning assembly language is in memorizing the different instructions, their effect, and their formats.

All of the instructions of the 6809 taken together are called the "instruction set" of the 6809. The instruction set of the 6809 is really the instruction set of the Color Computer as well, as no new instructions have been added by Radio Shack. Appendices I and II list the instruction set of the 6809.

Instruction Mnemonics

It's much easier to say ADDA than to say "Add the contents of the A register in the cpu together with a second operand and put the result in the A register." Lengthy instruction descriptions are simply denoted by an **instruction mnemonic.** There are two parts to the mnemonic, the "operation code," and the "operands."

Take a look at the program in Figure 1-1. This is a short 6809 assembly-language program to sort data. This program is shown just as it might have been entered to the Editor of EDTASM+, without having been assembled.

```
00100 * BUBBLE SORT
00110 BUBSRT  CLR   PASSNO     SET PASS # TO 0
00120 BUB010  LDX   #$400         POINT TO SCREEN
00130         LDY   #0            SET CHANGE FLAG TO 0
00140 BUB020  LDA   ,X+              GET FIRST ENTRY
00150         CMPA  ,X               TEST NEXT
00160         BLS   BUB030           GO IF A<=B
00170         LDB   ,X               GET SECOND ENTRY
00180         STB   -1,X             SWAP B TO A
00190         STA   ,X               SWAP A TO B
00200         LDY   #1               SET "CHANGE"
00210 BUB030  CMPX  #$400+511        TEST FOR SCREEN END
00220         BNE   BUB020           GO IF NOT ONE PASS
00230         INC   PASSNO        INCREMENT PASS #
00240         CMPY  #0            TEST CHANGE FLAG
00250         BNE   BUB010        GO IF CHANGE OCCURRED
00260 LOOP    JMP   LOOP       LOOP HERE
00270 PASSNO  FCB   0          PASS #
00280         END
```

Figure 1-1. Sample Color Computer Assembly-Language Program

You might want to enter the text below by using the EDTASM+ Edit mode, which works very similarly to BASIC Edit mode. Start by doing an Insert by entering I followed by ENTER and then entering the text a line at a time. After the last line, press BREAK, and you'll come back to the EDTASM+ "command mode." You can use the Edit mode "subcommands" just as in BASIC. (Refer to Appendix III.)

Hints and Kinks 1-2
EDTASM+ Edit Mode

The Edit mode in EDTASM+ is almost identical to the Edit mode in Extended Color BASIC. It works on a character by character basis within the line. Space along the line by space bar, backspace by left arrow. You can delete characters by D or nD, where n is the number of characters to be deleted. You can C(hange) characters by C, followed by one character or nC followed by "n" characters. There are many other commands, including H(ack) and K(ill) for those of you with a penchant for violence. See Appendix III for details.

Look at some of the text under the second column.

What you're seeing in the second column is the operation mnemonic for the instruction. This is sometimes called the **op code**, for "operation code." The op code LDY in the third line stands for Load the Y Register. The op code several lines down, the CMPA, stands for Compare.

The second part of the instruction mnemonic is the operands portion. This is sometimes called the argument portion. When taken together with the op code, the operand and op code define what the instruction does.

The operand for the LDY in the third line is #0, which, when taken together with the LDY, stands for "Load the Y register with a value of 0."

The CMPA line really means "Compare the contents of the A register with a memory byte at an address pointed to by the contents of the X register." (Quite a mouthful, eh? Don't worry about what the instructions do at this point. We'll be getting into that soon enough!)

Comment Lines

Any line that starts with an asterisk is simply a comment line and is not treated as an instruction. Any text in the last part of the line (the fourth column) is considered a comment, also.

Hints and Kinks 1-3
When Should You Use Comment Lines?

There can't be **enough** comment lines or comments after the operands. There is a special Hell for programmers who don't comment. They are doomed to figure out the Extended Color BASIC program by looking at machine-code values with faulty PEEKs.

In the code we're using in this book, you don't have to enter any comments if you're keying in the program yourself. They will not affect the program.

Labels

The BUBSRT, BUB010, and BUB020 names are called "labels." A label is used in assembly-language in lieu of a line number as in BASIC. They are equated to the memory location at which the instruction is stored. The LDY #0 instruction, for example, might be stored in RAM memory location $3F00, but the label BUBSRT would be the label of the instruction during assembly.

The Assembly-Language Process

The entire collection of text that defines 6809 instructions and comments constitutes an "assembly-language" program. What do we do with it? How do we feed it into the computer?

The 6809 microprocessor cannot accept text and decode it. Maybe the next generation of microprocessors will. We have to take the text representing the assembly-language program and convert it into a form that the 6809 can understand — binary ones and zeroes.

This process is called "assembly." EDTASM+ has a built-in assembler that will translate the text of the program in "assembly language" into "machine language," the binary ones and zeroes.

The EDTASM+ command for this is

```
A
```

for Assemble. The assembly command has different options, or "switches." The /IM "switch" means "assemble in memory rather than outputting code to cassette tape." Assuming that we had the text in memory, we could do a

```
*A/IM
```

to assemble the text.

Hints and Kinks 1-4 When Should You Use Assembler Output on Cassette?

Use the A command with "object" output to cassette only for 1) long programs for which the source file would take a long time to load 2) a relatively final version of the program.

It's so easy to load in a source file and reassemble, that outputting the assembler object to cassette is not as important as it could be. Of course, save the "source" or text file periodically, and make several copies. See the W command for saving source files.

After doing this you'd see a rapid display of the text in the assembly-language portion "scrolling up" on the screen. For the program above, you'll see something that looks like Figure 1-2.

```
                    00100 * BUBBLE SORT
0A6A 7F   0A95      00110 BUBSRT  CLR   PASSNO   SET PASS # TO 0
0A6D 8E   0400      00120 BUB010  LDX   #$400      POINT TO SCREEN
0A70 108E 0000      00130         LDY   #0         SET CHANGE FLAG TO 0
0A74 A6   80        00140 BUB020  LDA   ,X+          GET FIRST ENTRY
0A76 A1   84        00150         CMPA  ,X           TEST NEXT
0A78 23   0A        00160         BLS   BUB030       GO IF A<=B
0A7A E6   84        00170         LDB   ,X           GET SECOND ENTRY
0A7C E7   1F        00180         STB   -1,X         SWAP B TO A
0A7E A7   84        00190         STA   ,X           SWAP A TO B
0A80 108E 0001      00200         LDY   #1           SET "CHANGE"
0A84 8C   05FF      00210 BUB030  CMPX  #$400+511    TEST FOR SCREEN END
0A87 26   EB        00220         BNE   BUB020       GO IF NOT ONE PASS
0A89 7C   0A95      00230         INC   PASSNO     INCREMENT PASS #
0A8C 108C 0000      00240         CMPY  #0         TEST CHANGE FLAG
0A90 26   DB        00250         BNE   BUB010     GO IF CHANGE OCCURRED
0A92 7E   0A92      00260 LOOP    JMP   LOOP     LOOP HERE
0A95      00        00270 PASSNO  FCB   0        PASS #
          0000      00280         END
00000 TOTAL ERRORS

BUB010  0A6D
BUB020  0A74
BUB030  0A84
BUBSRT  0A6A
LOOP    0A92
PASSNO  0A95
```

Figure 1-2. Sample Assembly

Actually, we've "cleaned up" the display for you. The display on the Color Computer is in 2 lines because of the 32-character per line limitation of the

Color Computer, and we've taken it and put it on one line to make it easier to read.

The portion on the left of the text area represents the machine-language output of the assembler. This is sometimes called "object language" and the assembly-language text is sometimes called "source language."

The hexadecimal values in the second column are the actual hexadecimal codes, which, when converted to binary values, represent the codes for each instruction in the assembly-language program.

What have we really done at this point? We've really only used a program to translate a more English-like form of a program into binary ones and zeroes, the program being the assembler program in EDTASM+.

The machine-language form is on the screen. The actual machine-language values are also in memory at this point, and the memory locations in which they are stored are shown under the first column.

The first column shows a hexadecimal RAM location for the machine-language program. (We'll explain hexadecimal in the next chapter.) Note that the locations do not increment by one for each instruction. That's because the machine-language codes vary in size from 1 to 4 bytes in length. The BLS instruction, for example, is 2 bytes long in machine-language form. The LDX is 3 bytes long.

EDTASM+ is different from some assemblers that do not put the machine code in memory after assembly. In these assemblers, the "object code" goes onto a cassette or disk file. You can also do this with EDTASM+, by selecting another assembly option. We'll show you how in a later chapter.

Note that there is generally a "one-for-one" correspondence between an assembly-language form of the instruction, and the machine-language form of the instruction. One machine-language instruction is generated for each assembly-language instruction. Comment lines are ignored and do not generate any object code, along with certain other types of assembly-language text.

The A/IM command can be used to assemble any assembly-language program you have in EDTASM+. You may create your own program, using the EDTASM+ editing commands, or you can assembly existing assembly-language "source" programs that you've previously stored on cassette tape.

If you have a line printer on your system, you can use the

```
A/LP
```

form of the Assemble command to get an assembly-language **listing** of the program.

Another form of A is

A/IM/WE

This will assemble the program "in memory" and also halt the assembly if any assembly errors are found. The type of error will be displayed before the line in which the error occurred, as shown in Figure 1-3.

```
                     00100 * BUBBLE SORT FOR TWO-BYTE TABLE
0A1E 8E    0400      00110 BUBSRT  LDX     #$400    POINT TO SCREEN
0A21 7F    0A43      00120         CLR     CHANGE   RESET CHANGE FLAG
BAD OPCODE
00130 BUB011  L$$      ,X++       GET FIRST ENTRY
0A24 10A3 84         00140         CMPD    ,X         COMPARE
0A27 23    0D        00150         BLS     BUB021     GO IF A<=B
0A29 10AE 84         00160         LDY     ,X         GET 2ND ENTRY
0A2C ED    84        00170         STD     ,X         SWAP ENTRIES
0A2E 10AF 1E         00180         STY     -2,X
0A31 86    01        00190         LDA     #1         ONE
0A33 B7    0A43      00200         STA     CHANGE     SET CHANGE FLAG
0A36 8C    05FE      00210 BUB021  CMPX    #$400+510 TEST FOR END
UNDEFINED SYMBOL
BYTE OVERFLOW
0A39 26    FE        00220         BNE     BUB011     GO IF NOT ONE PASS
0A3B B6    0A43      00230         LDA     CHANGE   TEST CHANGE FLAG
0A3E 26    DE        00240         BNE     BUBSRT   GO IF CHANGE OCCURRED
0A40 7E    0A40      00250 LOOP    JMP     LOOP     LOOP HERE
0A43       00        00260 CHANGE  FCB     0        INITIALLY 0
           0000      00270         END
00003 TOTAL ERRORS

BUB011   0000     U
BUB021   0A36
BUBSRT   0A1E
CHANGE   0A43
LOOP     0A40
```

Figure 1-3. Assembly Errors

If you've never run an assembly before, take a moment now to enter the program and assemble it under EDTASM+. Use the A/IM or A/IM/LP assembly commands.

Executing (Running) the Progam

Once you have assembled a program you can execute it very easily in EDTASM+.

To execute any program under EDTASM+, get an error-free assembly listing by doing A/IM, go to ZBUG by entering a Z command, and then do a

GMMMM or GSTART

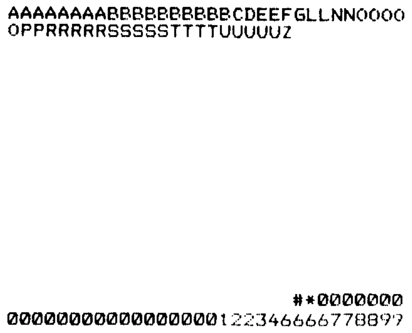
where MMMM is a hexadecimal address corresponding to the starting location of the program and START is the starting label.

Try it for this program. If you've entered the program correctly, the starting location of the program should be "BUBSRT," so do a

```
*A/IM
*Z
#XLOOP
#GBUBSRT
```

The X command sets a "breakpoint" in ZBUG. A breakpoint is simply a stopping point at which ZBUG regains control, and all we've done here is to tell ZBUG to execute the program starting at BUBSRT, but to regain control when the program reaches LOOP.

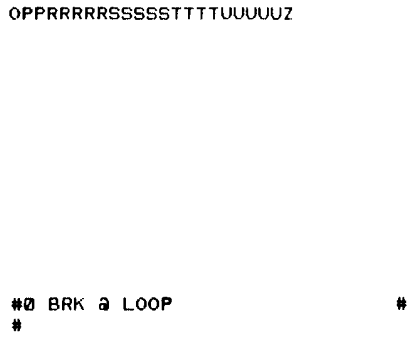
If you enter the sequence above, you should see the program execute. This program takes all of the text on the text screen in the Color Computer and "sorts" it in alphanumeric sequence. You should see all of the screen text rearrange itself in numeric order as shown in Figure 1-4. (Screen contents may differ from figure.)



```
AAAAAAAABBBBBBBBBBCDEEFGLLNNOOOO
OPPRRRRRSSSSSTTTTUUUUUZ









                           #*0000000
000000000000000000122346666778897
```

Figure 1-4. Sort Action

After the last part of the sort, the program will "fall through" to the breakpointed instruction as shown in Figure 1-5. (Screen contents may differ from figure.)



```
OPPRRRRRSSSSSTTTTUUUUUZ










#0 BRK a LOOP                    #
#
```

Figure 1-5. Breakpoint Action

EDTASM+ Edit Commands

Before we get into the discussion of actual instructions, let's mention some other related EDTASM+ commands.

- The Q command simply takes you back to BASIC. Any program or data you have in the EDTASM+ buffer will be lost; you'll have to reload EDTASM+ to continue.
- The P(rint) command is a misnomer that lets you display a range of lines from the text buffer on the screen. Use

  ```
  P#:*
  ```

  to print the entire text buffer or

  ```
  PLLL:MMM
  ```

  to print a range of lines from starting line LLL through ending line number MMM.

  The H(ardcopy) command lets you get a listing of the assembly-language portion of the text only (not the machine-language) on your system printer. The format for Hardcopy is

  ```
  HLLL:MMM
  ```

  where LLL is the starting line number and MMM is the ending line number. T is identical to the H format except that it prints text without line numbers.

  The N(umber) command renumbers the assembly-language lines. The format of N is

  ```
  NLLL,II
  ```

  where LLL is the starting line number for the renumber and II is the increment. Doing a

  ```
  N200,20
  ```

  would renumber the current lines with the new lines starting at line number 200 and increasing by 20 for each line.
- Another form of the I(nsert) command lets you insert lines between any existing lines. The format of Insert for this purpose is

  ```
  ILLL,II
  ```

  where LLL is the line number for the insert point and II is the increment. If you had text line 100, 110, 120, 130, and so forth, and you wanted three new lines between lines 120 and 130, you might say

  ```
  I121,1
  ```

The Insert mode would then be active and you'd see line 120 displayed in the text area, followed by the number 121. You could then enter the 3 new lines and press BREAK to exit. If you wished, you could then use the renumber command N to renumber.

If the Insert line number exists, the Insert mode will give an error message "NO ROOM BETWEEN LINES." Inserts can be done until the line numbers increment up to an existing line number. If you run out of space, simply renumber by N and continue the Insert.

The W command lets you Write an assembly-language "source" (text) file to cassette. The format of W is

```
W NAME
```

where NAME is the name of the file. Note that the file written is the assembly-language text only. It is not a file that can be run as a program. You can only use it to assemble under EDTASM+. If you don't provide a name for the W command, the name "NONAME" is used.

The L command lets you load a source file from cassette. The format of L is

```
L
```

or

```
L NAME
```

where NAME is an optional file name. If no name is specified, the next file on cassette will be loaded.

V(erify) works the same as load except that the file is not loaded but compared with the contents of the memory text buffer. It's used directly after a W(rite) to verify a good write.

You can switch back and forth between the Edit mode, assembler, and ZBUG at will as EDTASM+ contains the editor, assembler, and debug package in memory all at one time. This makes it very easy to edit, assemble, debug, and then go back to correct errors you've found by doing another edit of the text, reassembling, and trying it again.

Review

To recap what we've covered in this chapter:

- The instruction set of the 6809 includes dozens of instructions, some relating to arithmetic operations, and some relating to program control
- The instruction set is abbreviated by using mnemonics for the operation and the operands for each instruction

- Comment lines start with an asterisk and do not generate machine-language code
- Labels are used to represent an instruction by a symbolic form, rather than as an absolute location
- An assembler converts assembly-language code into the machine-language ones and zeroes that the 6809 requires for executing instructions
- The Q command returns you to BASIC or TRSDOS
- The P command allows display of text
- The H or T command allows hardcopy printing of the text
- The N command renumbers the text
- Insert lets you insert assembly-language lines between existing lines or to start a new program
- The W command lets you write out an assembly-language source file to cassette or disk; the V command lets you verify the write
- The L command allows you to load a source file previously written out by a W command

For Further Study

Appendix I 6809 Instructions Capsulized
Appendix III EDTASM+ Commands

KEY CHART — CHAPTER 2

INSTRUCTIONS

ABX	LBHS	CLRB	LDS	ROL
ADCA	BITA	CLR	LDU	RORA
ADCB	BITB	CMPA	LDX	RORB
ADDA	BLE	CMPB	LDY	ROR
ADDB	LBLE	CMPD	LEAS	RTI
ADDD	BLO	CMPS	LEAU	RTS
ANDA	LBLO	CMPU	LEAX	SBCA
ANDB	BLS	CMPX	LEAY	SBCB
ANDCC	LBLS	CMPY	LSLA	SEX
ASLA	BLT	COMA	LSLB	STA
ASLB	LBLT	COMB	LSL	STB
ASL	BMI	COM	LSRA	STD
ASRA	LBMI	CWAI	LSRB	STS
ASRB	BNE	DAA	LSR	STU
ASR	LBNE	DECA	MUL	STX
BCC	BPL	DECB	NEGA	STY
LBCC	LBPL	DEC	NEGB	SUBA
BCS	BRA	EORA	NEG	SUBB
LBCS	LBRA	EORB	NOP	SUBD
BEQ	BRN	EXG	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	JMP	PSHU	TFR
LBGT	LBVC	JSR	PULS	TSTA
BHI	BVS	LDA	PULU	TSTB
LBHI	LBVS	LDB	ROLA	TST
BHS	CLRA	LDD	ROLB	

ADDRESSING MODES

INHERENT
DIRECT
EXTENDED
IMMEDIATE
SIMPLE INDEXED
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	SETDP

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	R(EPLACE)
C(OPY)	~~L(OAD)~~	~~T(HARDCOPY)~~
D(ELETE)	M(OVE)	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
F(IND)	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	/NS NO SYMBOL TABLE
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	T DISPLAY BLOCK
B(YTE) MODE	T H HARDCOPY BLOCK
C(ONTINUE)	U MOVE BLOCK
D(ISPLAY)	V (ERIFY) BLOCK
E(DITOR)	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	Y (ANK) BREAKPOINT
I(NPUT) BASE	↑ **EXAMINE PRECEDING**
L(OAD) ML FILE	↓ **EXAMINE NEXT**
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
O(OUTPUT) BASE	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
R(EGISTER) DISPLAY	/ EXAMINE
S(YMBOLIC DISPLAY)	, SINGLE STEP

GENERAL TOPICS

CPU REGISTERS	SUBROUTINES
DATA TO REGISTERS	STACK OPERATIONS
LOADING AND STORING	ROTATES, SHIFTS
ADDITION AND SUBTRACTION	MULTIPLES
CONDITION CODES	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

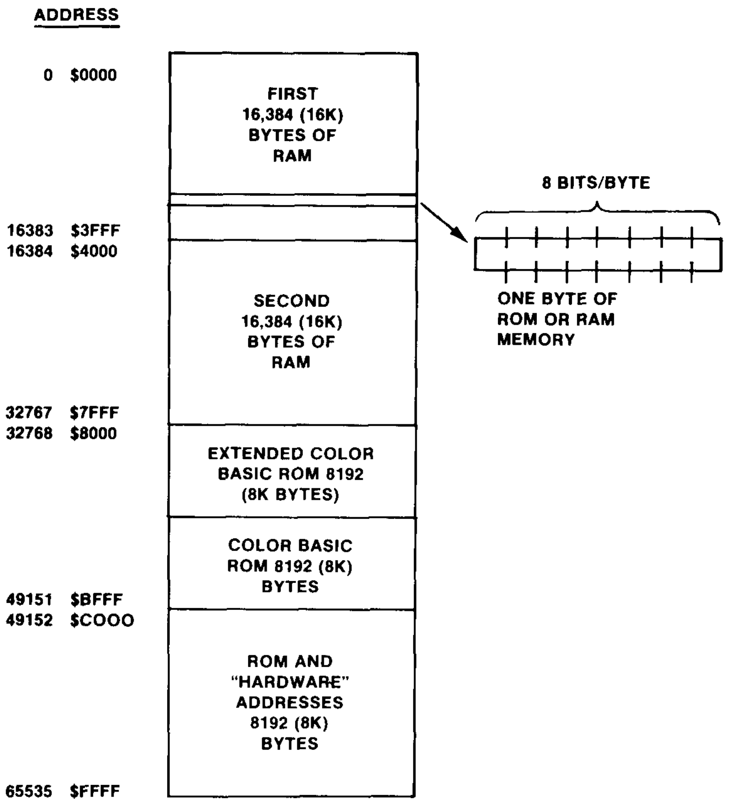
Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 2
6809 Registers

The 6809 has a number of high-speed memory locations within the microprocessor called "registers." We'll look at the "architecture" of the 6809 in this chapter and also get our feet wet with binary and hexadecimal operations in addition to looking at some ZBUG commands that will let you manipulate data.

What Is a 6809 Register?

By this time you probably know what RAM (random-access-memory) and ROM (read-only-memory) are, but we'll briefly describe them, anyway. RAM is a "read-write" memory organized in bytes, which are 8 **bits or binary digits** long, as shown in Figure 2-1. ROM is a "read-only" memory also organized in bytes, as shown in the figure.



ADDRESS
0 $0000
16383 $3FFF
16384 $4000
32767 $7FFF
32768 $8000
49151 $BFFF
49152 $COOO
65535 $FFFF
FIRST 16,384 (16K) BYTES OF RAM
SECOND 16,384 (16K) BYTES OF RAM
EXTENDED COLOR BASIC ROM 8192 (8K BYTES)
COLOR BASIC ROM 8192 (8K) BYTES
ROM AND "HARDWARE" ADDRESSES 8192 (8K) BYTES
8 BITS/BYTE
ONE BYTE OF ROM OR RAM MEMORY

Figure 2-1. Memory Bytes

A bit is either a 0 or a 1, on or off, lighted or unlighted. A typical byte of 8 bits, therefore, might be something like 10110111 — 8 different bits of any combination.

The 6809 microprocessor in the Color Computer contains a group of "registers," which are nothing more than memory locations, very similar to RAM.

Unlike RAM and ROM, though, which are addressed by a location value of 0 through 65,535, microprocessor registers are called by letter designations, as shown in Figure 2-2.

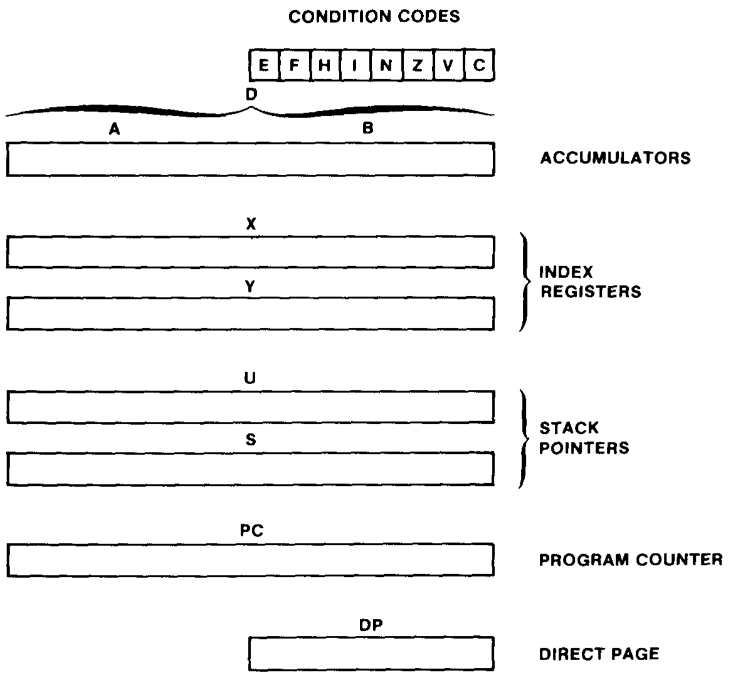


Figure 2-2. 6809 Registers

The letter designations are not necessarily related to their functions, although some are.

In general, 6809 registers, or "cpu registers," are used to hold the results of temporary operations. Because they can be read from or written to at faster speeds than RAM memory, using cpu registers rather than RAM memory locations speeds up the microprocessor instructions, which speeds up any program.

Hints and Kinks 2-1
What is a CPU?

CPU stands for "central processing unit." In fact, whenever we talk about cpu we mean the microprocessor in the Color Computer, the 6809. CPU is one of those archaic terms that dates back to the time of wooden computers and iron programmers.

The entire "instruction set" of the 6809 and other microprocessors is geared to using the cpu registers. Although there are many instructions which handle reading and writing to memory, just about all data passes through one or more of the cpu registers.

Register Functions

To examine the 6809 registers we'll have to use ZBUG, the "debugging" program of EDTASM+. Load EDTASM+ and enter a Z after the asterisk prompt. You'll automatically go to ZBUG as indicated by a pound sign prompt. Any time you see the # prompt, you'll know that you're in ZBUG. (To get back to the Editor/Assembler, enter an E.) Now enter an R. You should see this dialogue:

```
*Z
#R
 A=00 B=00 DP=00 CC=00 =
 X=0000 Y=0000 U=0000 S=0777
 PC=0000
#
```

This is a listing of all the 6809 cpu registers.

Hints and Kinks 2-2
When Do the Registers Change?

You'll notice that if you enter ZBUG "cold" that most of the registers are zeroed. The R command always displays the current state of the registers as modified by your program. You haven't done anything at this point, so all registers but S are zeroed. S points to a "stack area" in RAM, which we'll talk about in a later chapter.

All register contents are shown in **hexadecimal,** a shorthand way of representing binary. There are two hexadecimal digits for every 8 bits, so you'll see 2 hex digits under many of the registers.

Some of the registers hold 2 bytes instead of 1 byte however, twice as much as other registers or RAM locations. In this case you'll see four hex digits.

Hex digits are 0 through 9 and A through F. We'll explain them in a moment.

The last register is the PC, or Program Counter. This is the 2-byte register in the cpu that "points to" the next microprocessor instruction in memory.

First are "A" and "B." The A and B displays have 2 hex digits after the equals sign, as both the A and B registers hold 8 bits.

Next is the DP, or Direct Page, register. This is an 8-bit register represented by 2 hexadecimal digits. We'll talk about the DP in a later discussion, but for the time being we'll tell you that it allows the 6809 to reference memory locations by 256-byte pages, and is somewhat of an embellishment.

Next is a special register called "CC" or the condition codes register. This is really not a register at all, but a collection of 1-bit "flags" that are used to represent the results of instructions. These flags all have special meanings that we'll discuss in later chapters. For now, just note that there are 8 of them, and the 8 are represented by two hexadecimal digits.

The next four registers in the cpu are the X, Y, U, and S registers. Each of these are 16 bits or 2 bytes, and are represented by 4 hex digits.

Binary and Hexadecimal

If you want to do assembly-language programming, you'll have to become a little familiar with binary and hexadecimal notation. You won't have to become a math whiz at it, but you will have to at least be able to convert between decimal numbers and binary and hexadecimal and the reverse.

Let's consider 8- and 16-bit binary numbers, since those are the ones we'll be working with the most. Each 8-bit number represents data as shown in Figure 2-3.

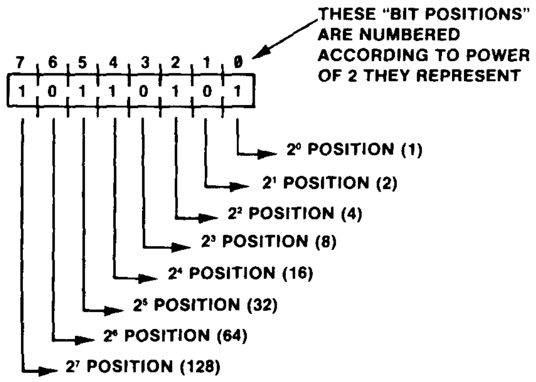


Figure 2-3. Binary Representation

Hints and Kinks 2-3 Must I Know Binary and Hexadecimal To Do Assembly Language?

Well, you can get by up to a point. . .uh. . .if you're not going to be doing much arithmetic. . .that is. . . .oh **all right**, the answer is YES. There goes another book sale. . .

On the other hand, you don't have to become adept at it immediately. Give it a chance, it's easier than it looks. Soon you'll be using binary and hex without much trouble.

Binary

Each digit position of an 8-bit binary number represents a power of two. The "bit position" on the right is 2 to the zero power, or 1. The next is 2 to the first power or 2. The next position is 2 to the second power, or 4. The next positions are 8, 16, 32, 64, and 128.

To find out the equivalent decimal number represented by an 8-bit binary number, just add together the powers of two represented:

```
10101100 =?
              128
              +32
               +8
               +4
              ---
10101100 =  172 decimal
```

To convert from decimal to binary, find out the powers of two that make up the decimal number. To convert 135 to binary, for example:

```
135 = ?
        Does 128 "go" into 135? - yes,        135-128=7
        Does 64 "go" into 7? - no
        Does 32 "go" into 7? - no
        Does 16 "go" into 7? - no
        Does 8 "go" into 7? - no
        Does 4 "go" into 7? - yes,            7-4=3
        Does 2 "go" into 3? - yes             3-2=1
        Does 1 "go" into 1? - yes             1-1=0
        135 = 128+4+2+1 = 10000111 in binary
```

If this all seems confusing, don't despair. We have an appendix to convert between decimal, binary, and hexadecimal in the back, Appendix IV. It will convert all decimal values from 0 through 255 into either binary or hex. We've also included a procedure to convert decimal values from 256 through 65,535 and that'll cover just about all of the numbers used in this book.

Before you look at the Appendix for binary conversions, though, try some values yourself:

- What is 01110010 binary in decimal?
- What is 10101010 binary in decimal?
- What is 255 decimal in binary?
- What is 33 decimal in binary?

Did you get the answers without looking in the Appendix? The answers are

- 01110010 = 114 decimal
- 10101010 = 170 decimal
- 11111111 = 255 decimal
- 00100001 = 33 decimal

When you're using 16-bit binary numbers, the process is the same, but the additional bits represent larger powers of 2, as shown in Figure 2-4. The "high-order" bits represent 32768, 16384, 8192, 4096, 2048, 1024, 512, and 256. The "low-order" 8 bits represent values as in 8-bit numbers. We won't burden you with a lot of exercises, but we'll just show you one conversion:

```
1010111100001111 = ?
                   32768
                    8192
                    2048
                    1024
                     512
                     256
                       8
                       4
                       2
                       1
                   -----
                   44815
```

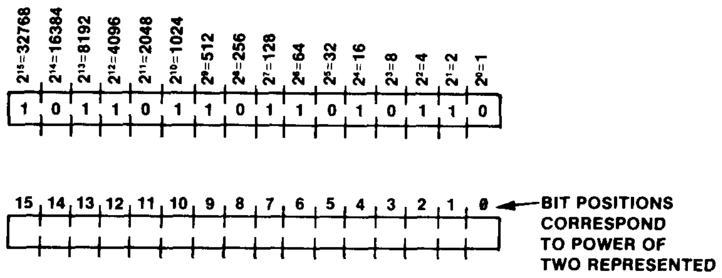


Figure 2-4. Sixteen-Bit Binary Numbers

You can see that it gets rather tedious to represent long strings of binary numbers. Programmers use a kind of shorthand to make things more compact. The shorthand is called "hexadecimal."

The hexadecimal numbers and their decimal equivalents for 0 through 15 are shown here

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0	9	1001	9
1	0001	1	10	1010	A
2	0010	2	11	1011	B
3	0011	3	12	1100	C
4	0100	4	13	1101	D
5	0101	5	14	1110	E
6	0110	6	15	1111	F
7	0111	7			
8	1000	8			

To convert any binary number to hexadecimal, simply divide into groups of 4 bits and use the chart on the previous page to find the hex equivalent of each group, as shown in Figure 2-5.

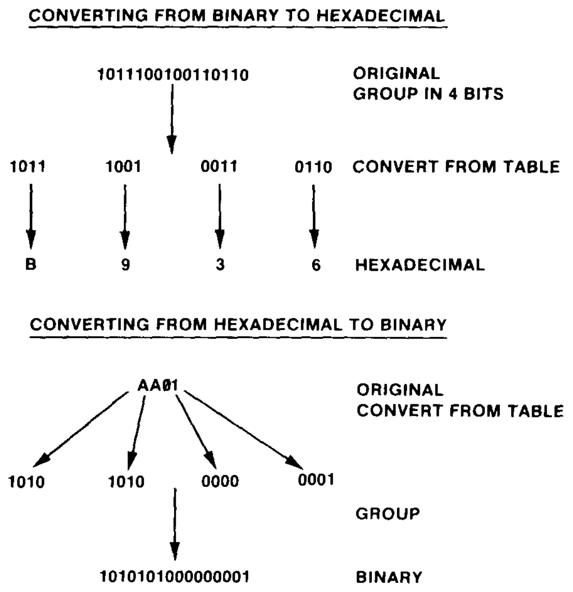


Figure 2-5. Converting from Binary to Hexadecimal

To convert back again, translate each hex digit to a 4-bit binary value, as illustrated in the figure.

You can also use the ZBUG "calculator mode" to easily convert between decimal and hexadecimal. Use this sequence to convert from decimal to hexadecimal:

```
#I10        (sets input "base" to decimal)
#O16        (sets output "base" to hexadecimal)
#16=10
```

To use the above sequence, get to ZBUG by typing Z from the Editor/Assembler. Then set the input "base" to decimal, or 10. Set the output "base" to hexadecimal, or 16. If you then type in any number followed by an equals sign, ZBUG will spew out the hexadecimal equivalent. Try some values.

Hints and Kinks 2-4
What is a "Base"?

The base of the number is the "power" used in the positional notation of a number. We use base 10 in decimal numbers — 9876 is actually 9 times 10 to the third power + 8 times 10 to the second power + 7 times 10 to the first power + 6 times 10 to the zero power.

The only reason we use base 10 is because we have 10 toes. (Anthropologists speculate that primitive man first counted on his toes and then went to his fingers when he saw more than 10 wildebeests.) An E.T. might use a base of 4 or 9, depending upon the number of digits he had.

To convert from hexadecimal to decimal, reverse the procedure:

```
#I16        (sets input base to hexadecimal)
#O10        (sets output base to decimal)
#16=22T
```

Notice that the "T" suffix indicates decimal. Any number typed in with an equals sign behind it will be taken to be a hexadecimal number by ZBUG, and ZBUG will return a decimal value. Try some values.

The easiest way to convert between the three types of numbers is to use ZBUG or the Appendix. After a while, you be able to work well with binary and hex numbers because you will see them frequently.

Hexadecimal numbers usually have a "$" prefix to indicate that they are in hex. The number $0100, for example, is 100 hex, or 256 decimal ($100=000100000000 binary).

Hints and Kinks 2-5
Representing Hexadecimal Numbers

Hexadecimal numbers can be represented by a prefix of "$" as in $0400 or a suffix of "H" as in 3000H in EDTASM+. We'll forget about the "H" suffix, as standard Motorola format for the 6809 is a prefix of "$."

Hexadecimal numbers sometimes have a "leading" 0 if the number starts with A through F. The reason for this is that many programs cannot decide whether a number like FACE is a hexadecimal number, or a text name. A 0FACE leaves no doubt.

Using the ZBUG Modify Register and Modify Memory Commands

To give you some experience in hexadecimal, we'll show you the ZBUG "open" register or memory commands. These commands allow you to observe the contents of a 6809 register or memory location and to change the contents. You can change the "input base" and "output base" just as we did in the calculator mode above.

ZBUG Modify Register

This command lets you look at the contents of a cpu register. The form is

```
register/
```

where "register" is a register name of A, B, DP, CC, X, Y, U, S, or PC and "/" is the slash key. For example, to look at the contents of A after entering ZBUG:

```
*Z
#A/      0                  (contents of A is 0)
```

To change the contents, just type in a value:

```
#A/      0       AA         (ENTER)
```

Here we've typed in hexadecimal AA ($AA). To see if the change was made, look at A again

```
#A/      0AA
```

Try some other values to see the changes.

In this case both the "input base" and "output base" were reset to hexadecimal on entering ZBUG. You can use the I and O ZBUG commands, though, to change either the input or output base or both. To input in decimal and display, or output, in hexadecimal, do

```
#I10
#O16
#A/      0AA     13         (A changed to 13 decimal)
#A/      0D                 (output is hexadecimal)
```

Try changing the other registers. Use the mnemonics of A, B, DP, CC, X, Y, U, S, and PC.

ZBUG - Modify Memory
The ZBUG slash operator also lets you display or change any location in memory. The format here is

```
#MMMM
```

where MMMM is a 1- to 4-digit hexadecimal value.

Enter

```
#3000/
```

You should have seen something like

```
#3000/ BITA <0FF
```

This is clearly not a decimal or hexadecimal value. In fact, it's a "mnemonic" value representing the instruction mnemonic at memory location hexadecimal $3000. To get a value, we've got to change the "examination mode" of ZBUG to "byte mode" by

```
#B
#3000/ 95
```

We'll talk about the mnemonic mode later, but for now, just remember to change to "byte" mode by a B if you expect to see the memory location displayed in hexadecimal.

Before you modify a memory location, let me give you this word of advice:

KNOW WHAT MEMORY LOCATION YOU'RE MODIFYING

If you don't know what memory location you're modifying, the results could be disastrous. BASIC and EDTASM+ use certain RAM locations for storing variables, and changing locations indiscriminantly could "blow up" BASIC or EDTASM+. Well, I said disastrous, but the worst that would happen is that you'd have to reset the Color Computer. All right, so it's not catastrophic ...Just remember, though, that if you start modifying low memory in the first 256 or 512 bytes you may see some unexpected results.

Try modifying RAM location $3000 by

```
#3000/ 95 123
#3000/ 23
```

Why'd we get the "23"? Well, the input mode was hexadecimal, and we entered 123, or hex $123, which is too large for the memory location. Remember, there are two hex digits for every 8 bits, and 4 hex digits for every 16 bits. EDTASM+ used the last two hex digits.

By the way, from now on I'll use "hex" as shorthand for "hexadecimal." I'll slip in a hexadecimal from time to time, however, just to remind you about the formal name.

You can use the down arrow key in ZBUG to examine or change a location and then examine the next location. As each location is examined, you can either enter a new value, or leave it as is. Just press DOWN ARROW after observing the value or entering a new value. Press ENTER to stop the sequence.

Suppose you wanted to change locations $3000 through $3005 to $12, $34, $56, $78, $9A, and $BC. You'd have something like

```
3000/  XX 12                (DOWN ARROW)
3001/  XX 34                     "
3002/  XX 56                     "
3003/  XX 78                     "
3004/  XX 9A                     "
3005/  XX BC                     "
```

(We've used the XX values to represent two hex digits the values of which may vary.)

To get out of the memory change mode, just hit ENTER before any value.

You can also use the UP ARROW in similar fashion to look at a preceding location. You can intermix UP ARROW and DOWN ARROW in any combination.

ZBUG will allow you to change any area of memory except the ROM area from location $8000 on up. Of course, you can't change non-existent memory, either! If you have a 16K machine and try to change location $7FFF, you won't have much luck

```
#7FFF/ 0FF          12          BAD MEMORY
#
```

Review

To review what we've learned in this chapter:

- A register is a special fast-access memory location in the microprocessor used to hold temporary results
- Registers are either one byte (8 bits) or two bytes (16 bits) long
- The registers are named A, B, DP, CC, X, Y, U, S, and PC
- Binary notation represents data with binary digits of 0 and 1
- Bit positions represent powers of two starting with 2 to the zero power (1) on the right and increasing powers to the left
- Numbers can be converted from binary to decimal by adding together the "weights" of 128, 64, 32, etc.
- Numbers may be converted from decimal to binary by seeing which powers of 2 "go" into the decimal number
- Hexadecimal representation is a shorthand notation for binary numbers
- Hexadecimal digits are 0 through 9 and A through F
- To change from binary to hex, convert 4-bit groups into hex digits; to reconvert, reverse the procedure
- The ZBUG slash command lets you change register values
- The ZBUG slash command also lets you display memory areas
- The input and output "bases" are controlled by the I and O commands in ZBUG. Use I10 or O10 for decimal and I16 or O16 for hexadecimal
- The ZBUG B command sets the "byte" examination mode

For Further Study

Appendix III EDTASM+ Commands
Appendix IV Binary/Decimal/Hexadecimal Conversions

KEY CHART — CHAPTER 3

INSTRUCTIONS

ABX	LBHS	**CLRB**	**LDS**	ROL
ADCA	BITA	CLR	**LDU**	RORA
ADCB	BITB	CMPA	**LDX**	RORB
ADDA	BLE	CMPB	**LDY**	ROR
ADDB	LBLE	CMPD	LEAS	RTI
ADDD	BLO	CMPS	LEAU	RTS
ANDA	LBLO	CMPU	LEAX	SBCA
ANDB	BLS	CMPX	LEAY	SBCB
ANDCC	LBLS	CMPY	LSLA	SEX
ASLA	BLT	COMA	LSLB	STA
ASLB	LBLT	COMB	LSL	STB
ASL	BMI	COM	LSRA	STD
ASRA	LBMI	CWAI	LSRB	STS
ASRB	BNE	DAA	LSR	STU
ASR	LBNE	DECA	MUL	STX
BCC	BPL	DECB	NEGA	STY
LBCC	LBPL	DEC	NEGB	SUBA
BCS	BRA	EORA	NEG	SUBB
LBCS	LBRA	EORB	NOP	SUBD
BEQ	BRN	**EXG**	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	JMP	PSHU	TFR
LBGT	LBVC	JSR	PULS	TSTA
BHI	BVS	**LDA**	PULU	TSTB
LBHI	LBVS	**LDB**	ROLA	TST
BHS	**CLRA**	**LDD**	ROLB	

ADDRESSING MODES

INHERENT
DIRECT
EXTENDED
IMMEDIATE
SIMPLE INDEXED
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	SETDP

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	**R(EPLACE)**
C(OPY)	~~L(OAD)~~	~~T(HARDCOPY)~~
D(ELETE)	**M(OVE)**	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
F(IND)	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	**/NS NO SYMBOL TABLE**
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	T DISPLAY BLOCK
B(YTE) MODE	T H HARDCOPY BLOCK
C(ONTINUE)	U MOVE BLOCK
D(ISPLAY)	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	**Y (ANK) BREAKPOINT**
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	**/ EXAMINE**
S(YMBOLIC DISPLAY)	, SINGLE STEP

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
DATA TO REGISTERS	STACK OPERATIONS
LOADING AND STORING	ROTATES, SHIFTS
ADDITION AND SUBTRACTION	MULTIPLES
CONDITION CODES	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type = **Present Chapter**
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 3
Transferring Data to Registers

In this chapter we'll actually start using assembly language. The cpu registers are like other memory locations in that they can be "loaded" with data values. We'll find out how here. We'll also discuss how to move data between the various cpu registers. All in all, not as exciting as a Steven Spielberg movie, but a close second.

First of all, you might glance back at Chapter 2 to review the register "architecture," a fancy word for describing what registers are available for the assembly-language programmer and what their chief uses are.

The "general-purpose" cpu registers are the A and B registers. They're used to manipulate 8 bits of data at a time. Both the A and B registers have equal status — anything you can do with the A register you can also do with the B register. The A and B registers are "accumulator" registers. This is a somewhat archaic term that means that these registers are used to accumulate results of adds, subtracts, and other operations.

Remember that the A and B registers can be grouped together to make up a 16-bit register called the "D" register. When grouped this way, the A register is the "most significant" or left-hand 8 bits, and the B register is the "least significant" or right-hand 8 bits, as shown in Figure 3-1.

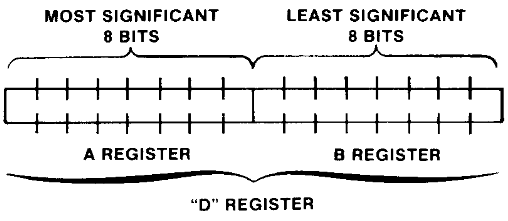


Figure 3-1. D Register

The "D" register can be used for 16-bit arithmetic operations such as adds and subtracts, but you can't do as much with the D register as you can with either the A or B registers.

The remaining registers are not "general-purpose" registers, but are used for stack operations (S and U), indexing operations (X and Y), program control (PC), or setting the "page" address (DP).

Loading Registers

Let's first look at simple "loads" of the A and B accumulators. Look at the code below. We'd recommend entering each of these sets of code as we discuss the operations, and we've geared the examples to let you see what's happening. There's no need to enter the comments after the semicolon, as we've included them just to explain the operations, although you can if you wish.

Hints and Kinks 3-1
More EDTASM+ Edit Commands

There are a number of other Editor commands that we should cover here, although they are also in the EDTASM+ manual.

The R(eplace) command works somewhat like Insert, but it replaces an existing line with a new line. Use it to overwrite existing lines when entering "source code."

The D(elete) command deletes a range of lines from the text buffer.

D100:300

for example, deletes lines 100 through 300.

The C(opy) command copies a range of lines to a new location with a new starting line number and increment.

C1000,100:300,10

for example, copies the range of lines from 100 through 300 to a location starting with 1000 and with increment 10. The original block of lines is left unchanged. M(ove) works exactly like C(opy) except that the original lines are deleted.

F(ind) can be used to find a character string somewhere in the text buffer. Suppose that you had a label "NEXT" but couldn't find it in displaying the buffer. You could use

FNEXT

to search the text buffer for the string. The string may be any number of characters. If you use F alone, the last defined string will be used.

```
                      00100 * LOAD THE REGISTERS
092E 86    37         00110 START    LDA    #55      LOAD A WITH 55
0930 C6    55         00120          LDB    #$55     LOAD B WITH 85
0932 8E    03E8       00130          LDX    #1000    LOAD X WITH 1000
0935 108E  1234       00140          LDY    #$1234   LOAD Y WITH 4660
0939 10CE  3F00       00150          LDS    #$3F00   LOAD S WITH $3F00
093D CE    3000       00160          LDU    #$3000   LOAD U WITH $3000
0940 CC    03E8       00170 ENDEX    LDD    #1000    LOAD A,B WITH 1000
0943 7E    0943       00180 LOOP     JMP    LOOP     LOOP HERE
           0000       00190          END
00000 TOTAL ERRORS

ENDEX    0940
LOOP     0943
START    092E
```

Figure 3-2. Load Registers Program

Assemble by entering A/IM and check for errors. If you have a "clean" assembly, you can execute by doing the following:

```
*Z          (go to ZBUG)
#XLOOP      (set breakpoint at LOOP instruction)
#GSTART     (execute from START)
```

If you follow this sequence, you'll see that ZBUG "hits" the breakpoint after executing each of the instructions and responds with

```
#XLOOP      (setup breakpoint)
#GSTART     (execute)
 0          BRK @ LOOP
 #
```

What this means is that the 6809 has executed each of the instructions in this short program and then reached the breakpointed instruction at location LOOP. **THIS INSTRUCTION WAS NOT EXECUTED** but control was turned back to ZBUG.

After you reach the breakpoint, look at the contents of the 6809 registers by doing an R command in ZBUG. You'll see

```
#R
 A=37 B=55 DP=00 CC=80 =E
 X=03E8 Y=1234 U=3000 S=3F00
 PC=094D                          (PC may be different)
 #
```

Hints and Kinks 3-2
Why the PC Register Varies

The PC register may be a different value on your system than it is in the examples in this book. EDTASM+ "assembles" your program directly after a text buffer that holds your source code. If your source code is not exactly equal in length to the examples in this book, the start of the program may vary. If, for example, you haven't included comments, the program start will be much lower than the examples in this book. The PC points to the breakpoint location after the breakpoint occurs, and this also moves in step with the program start, as the length of the program is usually fixed.

Hints and Kinks 3-3
What are the Symbols at the End of the Assembly?

Every time you assemble, EDTASM+ compiles a list of symbols found in the source code. The symbols are generally "labels" in the first column of the source code. In a proper assembly, all symbols must appear somewhere as a label. The symbols are put into a symbol "table," along with their corresponding location value. The start of your program might be labeled "START" for example, and when your program is assembled, the instruction at START might be at "absolute" location $0990. At the end of the assembly, the entire symbol table is listed, so that you can see where in memory the instructions associated with the labels reside. You can delete the symbol table listing by doing an assembly of the form A/IM/NS, where the /NS "switch" stands for "No Symbol Table."

Each of the 6 registers has been loaded with a data value. Look at the A register, for example. In the source code line, we had an "LDA #55" instruction, standing for "Load A with 55." If you look at the A= display, you'll see 37, which is the hexadecimal equivalent for decimal 55.

What about the B register? We loaded it with a $55. Remember that the "$" is a special hexadecimal prefix which says that the number for the load is in hexadecimal in the source code line. Sure enough, if we look at the B display, we'll see a 55.

Look at the display for X, Y, S, and U. Do they correspond to the values that we specified in the source lines? Well, we loaded X with 1000 decimal, which is $3E8 in hexadecimal. We loaded Y with $1234, which is already a hexadecimal value and displayed as such. We loaded S with $3F00 and U with $3000, and they appear to be loaded properly also.

When you ran this program by the G command, the six instructions were executed in about 20 microseconds (a microsecond is a millionth of a second) and loaded the A, B, X, Y, S, and U registers with the values shown. Assembly language is so fast that it's hard to believe that anything actually happened, but the ZBUG R command displays the actual values in the 6809 registers after the program.

By the way, what is the largest number that can be held in 8 bits? In other words, what is the largest number that can be loaded into an 8-bit general register? We've loaded the B register with this value, which is a decimal 255, or a hexadecimal $FF.

Immediate Loads

This type of instruction used an "addressing" mode called "immediate" addressing. In his type of addressing, the data for the instruction is contained in the instruction itself. Look at the assembly listing for the LDA #$XX, for example.

In this case we wanted to load 55 into the A register. The value of 55 decimal is (as you've figured out) $37, and if you look in the machine-language data for the LDA #55, you'll see a $86 byte, followed by an $37. In fact, all of these instructions have the "immediate" data to be loaded in their second byte or second or third bytes. If we're loading a 16-bit register, such as X, we'll see two bytes of immediate data. Look at the LDX $1000 instruction as an example. You'll see a $8E byte, followed by $03 and $E8. The $03,$E8 is a 16-bit value of $03E8, or decimal 1000.

Anytime that you see an LDA, LDB, LDD, LDS, LDU, LDX, or LDY in the source line, followed by a value in decimal or hexadecimal **with a leading "#",** the LD will be an "immediate 8-bit or 16-bit load" that loads data into the register.

Sharp-eyed "hackers" among you may have noticed that when we breakpointed, we did not execute the next-to-last instruction of our short program, the LDD #1000. You can execute this by breakpointing at the last instruction of the program, the LOOP JMP LOOP. This is an instruction that jumps to itself, much like a 100 GOTO 100 in BASIC. The sequence for executing this instruction is

```
#Y          (reset all breakpoints)
#XLOOP      (setup breakpoint)
#GSTART     (execute)
```

After the breakpoint is reached, the D register will be loaded with 1000. What will you see? Since the D register is A and B combined, you'll see A set to $03 and B set to $E8. You can verify this by doing an R command in ZBUG.

Hints and Kinks 3-4
More On Breakpoints

You can have up to 8 breakpoints in ZBUG. They're numbered 0 through 7 so that you can keep track of where they are. To see which breakpoints you've used, enter the D command in ZBUG. You'll get something like:

```
#D
 0  BRK @ START
 1  BRK @ END
```

To reset breakpoints, use the Y command. The Y command was created by a British programmer at Microsoft and stands for "Yank." Picture the breakpoints being yanked out of the breakpoint table, and you'll remember the mnemonic. The Y command alone resets **all** breakpoints, but Y with a number resets a specific breakpoint and leaves the others set. Y3, for example, yanks breakpoint 3 but leaves the others.

Transferring Data Between CPU Registers

We've seen how to load 8- and 16-bit registers with immediate data, but what about "moving" data between cpu registers? This is done all the time in assembly-language programs. Often we want to move data from the A register into the B register and back again or from the D register into the Y register or some other combination.

In this case we use the "TFR" instruction. This time we won't be loading immediate data, but "transferring" or copying the contents of one register into another.

If you want to see how this works, delete the source code of the short program above by doing

```
#E          (back to Editor from ZBUG)
*D#:*       (deletes from beginning to end)
```

You can now enter this short 6809 source code segment:

```
                        00100 *LOAD BETWEEN REGISTERS
0925 86    55           00110 LAST    LDA     #$55     IMMEDIATE LOAD OF $55
0927 C6    00           00120         LDB     #0       IMMEDIATE LOAD OF 0
0929 1F    89           00130         TFR     A,B      TRANSFER A TO B
092B 8E    1234         00140         LDX     #$1234   IMMEDIATE LOAD OF $1234
092E 108E 0000          00150         LDY     #0       IMMEDIATE LOAD OF 0
0932 1F    12           00160         TFR     X,Y      TRANSFER X TO Y
0934 1F    01           00170         TFR     D,X      TRANSFER D TO Y
0936 7E    0936         00180 LOOP    JMP     LOOP     LOOP HERE
           0000         00190         END
00000 TOTAL ERRORS

LAST      0925
LOOP      0936
```

Figure 3-3. Load Between Registers Program

This program shows you how the registers can transfer data to each other by the TFR instruction. Enter it, edit it until it assembles properly, breakpoint at LOOP, execute by GLAST, and then look at the register contents with the ZBUG R command.

Here's what the program did: The A register was first loaded with hexadecimal 55. The B register was then cleared by loading it with 0. Next, a TFR transferred the contents of A to B. A and B both contain $55 at this point. Next, the X register was loaded with hexadecimal 1234. Next, the Y register was loaded with 0. The TFR instruction then transferred the contents of X to Y. Both X and Y contain $1234 at this point. The last instruction then transfers the contents of D (A and B) to X. Since A and B each hold $55, the result in X is $5555.

The A or B register can be loaded by the other accumulator (or DP) simply by using the TFR instruction. In this instruction, the "source" register is first, followed by the "destination", as in TFR A,B, which loads B with the contents of A. The same thing holds true for 16-bit registers. TFR X,Y transfers the contents of X into Y. An important point: You can't load a 16-bit register with an 8-bit register and vice versa. You couldn't say

```
TFR     A,Y
```

You'd get a REGISTER ERROR message on assembly.

The EXG Instruction

The TFR instruction **copies** the contents of one cpu register to another cpu register. The EXG, however, **exchanges** the contents of one cpu register with another.

```
EXG     A,B
```

for example, exchanges A with B, and

```
EXG     X,Y
```

exchanges X with Y.

A Special Clear Instruction

There's a special instruction to load a 0 data value into A or B, the CLRA or CLRB instructions. As you might guess, "CLR" stands for "clear," and the instructions are the same as

```
LDA     #0      (use CLRA instead)
LDB     #0      (use CLRB instead)
```

As "good programming practice," you should use CLRA and CLRB in place of the LDA or LDB, as it is a 1-byte instruction and executes more rapidly.

Hints and Kinks 3-5
Is It Important to Use Efficient Instructions?

It depends. Your programs might run 0.001% faster by using CLRAs instead of LDA #0s. However, if you try to use the most efficient instructions for **every** type of instruction, your programs might run 20% faster in some cases, or even more. There'll come a time when you want to squeeze every last bit of speed out of your programs, and good assembly-language habits learned early will pay off. On the other side of the coin, assembly language is so much faster than BASIC you can be very "sloppy" and still get by nicely.

Hints and Kinks 3-6
Stopping the Assembly

You can stop the assembly listing in mid assembly by pressing the SHIFT and @ keys at the same time. Continue the listing by pressing any key.

Inherent Addressing

Instructions such as CLRA and CLRB assemble as a single byte and always have the same format. The "Addressing mode" for these simple instructions is called "Inherent addressing," as the instructions are "self-contained" and don't require operands. See Appendix II for other instructions of this type.

Review

To recap what we've learned in this chapter:

- You can load any 6809 register except the DP register by an "immediate" load

- You can load A with B or B with A or one 16-bit register with another 16-bit register by a TFR of the form TFR S ,D, where S is "source" register, and "D" is "destination" register
- The CLRA or CLRB clears the A or B accumulators
- Inherent addressing is used for instructions that are fixed format and do not require operands

For Further Study

LD 8-bit and 16-bit immediate instructions (see Appendix II)
TFR instruction
CLRA, CLRB instructions

KEY CHART — CHAPTER 4

INSTRUCTIONS

ABX	LBHS	~~CLRB~~	~~LDS~~	ROL
ADCA	BITA	CLR	~~LDU~~	RORA
ADCB	BITB	CMPA	~~LDX~~	RORB
ADDA	BLE	CMPB	~~LDY~~	ROR
ADDB	LBLE	CMPD	LEAS	RTI
ADDD	BLO	CMPS	LEAU	RTS
ANDA	LBLO	CMPU	LEAX	SBCA
ANDB	BLS	CMPX	LEAY	SBCB
ANDCC	LBLS	CMPY	LSLA	SEX
ASLA	BLT	COMA	LSLB	STA
ASLB	LBLT	COMB	LSL	STB
ASL	BMI	COM	LSRA	STD
ASRA	LBMI	CWAI	LSRB	STS
ASRB	BNE	DAA	LSR	STU
ASR	LBNE	DECA	MUL	STX
BCC	BPL	DECB	NEGA	STY
LBCC	LBPL	DEC	NEGB	SUBA
BCS	BRA	EORA	NEG	SUBB
LBCS	LBRA	EORB	NOP	SUBD
BEQ	BRN	~~EXG~~	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	JMP	PSHU	TFR
LBGT	LBVC	JSR	PULS	TSTA
BHI	BVS	~~LDA~~	PULU	TSTB
LBHI	LBVS	~~LDB~~	ROLA	TST
BHS	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
DIRECT
EXTENDED
~~IMMEDIATE~~
SIMPLE INDEXED
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	**SETDP**

Bold Type = Present Chapter
Regular Type - Future Chapters
~~Italic Type~~ = *Past Chapters*

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	T DISPLAY BLOCK
~~B(YTE) MODE~~	T H HARDCOPY BLOCK
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	, SINGLE STEP

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
LOADING AND STORING	ROTATES, SHIFTS
ADDITION AND SUBTRACTION	MULTIPLES
CONDITION CODES	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Chapter 4
Loading and Storing Data Between Registers and Memory

In this chapter we'll see how data can be loaded into cpu registers **from** memory. This operation is called a **load** just as it was in the case of immediate loads in the last chapter. We'll also see how data can be transferred from a cpu register **to** memory. This operation is called a **store.** There are many **addressing modes** that can be used with loads and stores, and we'll discuss the simpler ones here.

Extended Addressing

The simplest way of loading a register from memory or storing a register into memory is by **extended addressing.** In this addressing mode the address of the memory location involved is specified in the instruction itself. Look at the following code to see what we mean:

```
                     ØØ1ØØ *LOADS AND STORES BY EXTENDED ADDRESSING
Ø915 86   2D         ØØ11Ø START   LDA     #45      LOAD A WITH 45
Ø917 B7   3ØØØ       ØØ12Ø         STA     $3ØØØ    STORE IN $3ØØØ
Ø91A F6   3ØØØ       ØØ13Ø         LDB     $3ØØØ    LOAD B WITH ($3ØØØ)
Ø91D 8E   Ø3E8       ØØ14Ø         LDX     #1ØØØ    LOAD X WITH 1ØØØ
Ø92Ø BF   3ØØ1       ØØ15Ø         STX     $3ØØ1    STORE IN $3ØØ1
Ø923 1ØBE 3ØØ1       ØØ16Ø         LDY     $3ØØ1    LOAD Y WITH ($3ØØ1)
Ø927 7E   Ø927       ØØ17Ø LOOP    JMP     LOOP     LOOP HERE
          ØØØØ       ØØ18Ø         END
ØØØØØ TOTAL ERRORS

LOOP     Ø927
START    Ø915
```

Figure 4-1. Loads and Stores, Extended Addressing Program

Use the Edit mode of EDTASM+ to enter the source code above. After entering, you can assemble with A/IM. Check for errors, and if there are none, go to ZBUG. The dialogue will go something like this:

```
*I            (Insert mode in EDTASM+)
              (enter source code)
*A/IM         (Assemble source code)
*Z            (go to ZBUG)
#             (ZBUG prompt)
```

You're now ready to execute the program. Breakpoint at location LOOP and do an execute from location START. You'll see an almost instantaneous execution as the breakpoint is reached. The sequence will look like this:

```
#XLOOP        (breakpoint at location LOOP)
#GSTART       (execute from location START)
 0 BRK @ LOOP
#
```

Let's see what the program on the previous page did. The LDA #45 instruction uses immediate addressing, as we discussed last chapter. It loads a value of decimal 45 into the A register.

The STA $3000 is a "store" that takes the contents of the A register and stores it into a memory location, in this case the memory location at $3000. Since the contents of A was decimal 45, a 45 is stored into memory location $3000.

To see that this is true, you can examine location $3000 by:

```
        #B                      (set Byte mode)
        #3000/        2D
```

The next instruction, LDB $3000, loads the B register not with immediate data, but with the **contents** of memory location $3000. Since we've just stored 45 there, a 45 will be loaded into the B register. You can see that result by using the ZBUG R command to look at the registers.

The LDX #1000 instruction is an immediate load of decimal 1000 into the X register. The value of 1000 decimal in binary is $03, $E8 in hex, and you'll see this value in X if you use the R command.

The STX $3001 stores the contents of X into memory locations $3001 and $3002. Note that this store is a two-byte store of the memory location specified, plus the next location, in this case $3001 and $3002. If you examine $3001 and $3002 you'll see a $03 in $3001 and $E8 in $3002.

```
        #3001/        03       (DOWN ARROW)
         3001/        0E8
```

The next instruction, LDY $3001, loads the contents of $3001 and $3002 into the Y register. Again, this is a 16-bit load because the Y register is 16 bits, and it takes the contents of the memory address specified plus the contents of the next memory address for the load. Since we just stored $03, $E8 into $3001 and $3002, the same two values are loaded into Y.

Loads and Stores between cpu registers and memory, then, are really pretty simple. They simply transfer 8 or 16 bits of data between a cpu register and memory. If the register involved is 8 bits, 8 bits are transferred; if the register involved is 16 bits, 16 bits are transferred. The data is **copied** and the original data in the register or memory location(s) is not destroyed.

There are loads and stores for all cpu registers except for the Direct Page (DP) and Program Counter (PC):

Loads:
LDA, LDB, LDD, LDS, LDU, LDX, LDY

Stores:
STA, STB, STD, STS, STU, STX, STY

Note that the LDD and STD operate with D, the 16-bit register made up of A (most significant 8 bits) and B (least significant 8 bits).

Now let's take a look at the instruction format itself. If you assemble the code above and look at the STA $3000, you'll see a $B7 byte followed by $30 and $00. The $B7 byte is the "opcode" of the instruction. It marks the instruction as an "STA." The second and third bytes of the instruction are the memory address value to be used in the STA, in this case $3000. As there are two bytes, memory address values of $0000 through $FFFF can be used, although some of these are ROM locations.

The data in the second and third bytes was not "immediate" data as in the case of the immediate instructions, but an address value. This type of addressing is called "extended addressing" as it allows loads or stores or other operations using any of the 65,536 bytes of memory in the 6809 addressing range.

Hints and Kinks 4-1
Instruction Formats

The standard instruction format is one or two bytes of "opcode" followed by other bytes relating to operand address in memory, immediate value, or other "argument."

The opcode term stands for "operation code." It is a literal code value that the 6809 decodes to determine what type of instruction is being executed, how many operands will be involved, and other specifics pertaining to the instruction.

The 6809 is "optimized" in regard to instruction length. The more frequently an instruction is used, the shorter it is. This means that more frequently used instructions, such as LDA, will have one byte of opcode, while less frequently used instructions, such as LDY, will have 2 opcode bytes.

Look at the LDY $3001 instruction. In this case there's a $10 followed by $BE and then a byte of $30 and one of $01. The first two bytes in this case are the "opcode," marking the instruction as an LDY, and the third and fourth bytes are the address for the Load.

Anytime that the immediate # prefix is not used in the operands column of the source code, the load or store will take the operand from memory rather than from an immediate value in the instruction. This "extended addressing" is used in many different types of instructions.

Simple Indexed Addressing

Now let's look at another type of addressing, called "indexed addressing." We'll cover a simple case of it here, and expand upon this addressing mode in another chapter.

A rudimentary form of indexed addressing uses the X and Y registers. X and Y were designed for this purpose and were made 16 bits "wide" so that they could hold memory address values of $0000 through $FFFF.

In the simplest case of indexed addressing, X or Y "points to" a memory address and acts as the address for the instruction. Suppose that we had the following code:

```
                        00100 *LOADS AND STORES USING SIMPLE INDEXING
090A 8E    3000         00110 START    LDX     #$3000  POINT TO $3000
090D 108E  3001         00120          LDY     #$3001  POINT TO $3001
0911 86    2D           00130          LDA     #45     LOAD A WITH 45
0913 C6    0B           00140          LDB     #11     LOAD B WITH 11
0915 A7    84           00150          STA     ,X      STORE A INTO $3000
0917 E7    A4           00160          STB     ,Y      STORE B INTO $3001
0919 7E    0919         00170 LOOP     JMP     LOOP    LOOP HERE
           0000         00180          END
00000 TOTAL ERRORS

LOOP     0919
START    090A
```

Figure 4-2. Loads and Stores, Simple Indexing Program

The LDX #3000 loads the X register with an immediate value of $3000. This immediate value is actually a memory address to be used in an "indexing" operation. The Y register is then loaded with an immediate value of $3001. Now A and B are loaded with 45 decimal and 11 decimal, respectively.

The next two instructions perform "indexed" operations. The STA ,X stores the contents of A into the memory location pointed to by the contents of X. X contains a $3000, so the instruction is the same as

STA $3000

The STB ,Y stores the contents of B into the memory location pointed to by the Y register in a second indexing operation. Here the instruction is the same as

STB $3001

as Y contained a $3001.

Why not just use an "extended addressing" STA and STB instead of going to all that trouble? It's true that the X and Y registers have to be loaded with a "pointer value," and that in this case the indexed operations take more instructions than just doing an STA or STB. However, the pointer value in X and Y can be adjusted quite easily once it is initialized. X and Y can be used to point to a block of data to be processed and can then be adjusted to make the processing operation more efficient than doing "extended addressing" operations. We'll discuss more about indexed operations in later chapters. For now, however, just remember that anytime you see the format

,X or ,Y

it means "use the contents of X and Y as a pointer value to a memory location."

Hints and Kinks 4-2
Using Other Registers as Index Registers

If you promise not to tell this to any other reader, we'll jump the gun here and tell you that the 6809 is so powerful as far as addressing modes that not only can X and Y be used as index registers, but so can U and S. The 6800 predecessor to the 6809 had only one index register, X. The 6809 adds three more, Y, U, and S. The X and Y are registers are typically used as index registers, while S and U are used typically used as "stack pointers." More on all these topics in later chapters.

Direct Page Addressing

Direct Page addressing, as you might have guessed, uses the DP register. When the Color Computer and 6809 are powered up, this DP is automatically set to $00. The DP register cannot be loaded directly, but can be loaded by the TFR instruction we discussed in the last chapter. To load the DP register with $12, for example, you'd do a:

```
        LDA #$12      LOAD A WITH $12
        TFR A,DP      LOAD DP WITH A ($12)
```

Try this experiment: Enter the following program into EDTASM+

```
START   LDA $50       LOAD A WITH LOCATION $0050
        END
```

Now assemble and look at the machine-language bytes assembled for the instruction. What did you see?

You should have seen a $96 byte followed by a $50 byte. Wait a minute, though! Shouldn't this have been an "extended addressing" instruction?. It wasn't an immediate load, but a load of memory address $50 into A.

In fact, this instruction was assembled as a "direct addressing" type of instruction. The direct addressing mode takes the current contents of the DP register as the high-order 8 bits of the address and uses the operand in the instruction as the low-order 8 bits. In this case, since DP=0, the total address would be $0050. The resulting address, or "effective address" would be used in the load just as if (in this case) the instruction would have been an "extended addressing" type.

The direct mode saves one byte over the extended addressing type as it requires only one byte of the address in the instruction. The remaining byte is contained in the DP register. Figure 4-3 shows the process.

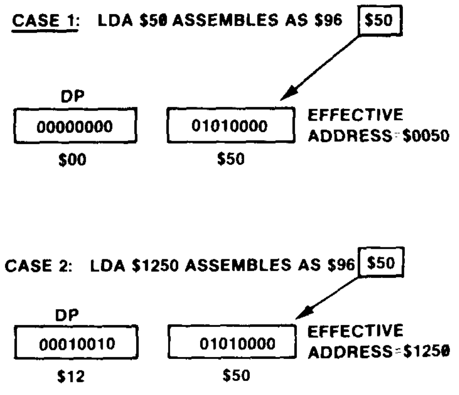


Figure 4-3. Direct Addressing

In this addressing mode DP always points to the beginning of a 256-byte page. If DP is never loaded, it remains pointing to "page 0" as it contains $00. If DP is loaded with any new value, then it points to the beginning of a 256-byte page. Suppose DP were loaded with $12 by the sequence above. It would then point to $1200 and instructions that referenced locations $1200 through $12FF could use the 2-byte direct address mode in place of the extended addressing mode (see Figure 4-3).

Not only does the direct addressing mode save one byte over extended, but it's about 20% faster besides.

Now try another experiment. Load the DP register with $30, so that instructions that referenced locations in the $3000 through $30FF area could use direct addressing:

```
START     LDA    #$30      LOAD A WITH $30
          TFR    A,DP      LOAD DP WITH $30
          LDA    $3001     LOAD A WITH ($3001)
          END
```

By rights the LDA $3001 should be assembled as a direct addressing type of instruction with an "opcode" byte of $96 and a "low-order" address byte of $01. When we assemble, though, we see this for LDA:

```
B6 3001
```

indicating that the instruction actually assembled as an "extended addressing" LDA! Why?

The reason is that you knew that the DP register had $30 in it, and I knew that the DP register has $30 in it, but the assembler in EDTASM+ did not! It must be told via a "pseudo-op," a command to the assembler that is not an instruction. The SETDP pseudo-op lets the assembler in on the contents of DP:

```
START      LDA    #$30       LOAD A WITH $30
           TFR    A,DP       LOAD DP WITH $30
           SETDP $30         SET ASSEMBLER
           LDA    $3001      LOAD A WITH ($3001)
           END
```

Assembling the code above results in

```
96 01
```

for the LDA $3001, which uses the direct addressing mode.

The SETDP operand can be any number from $00 through $FF to match what is in the DP. (In special cases, it does not have to match the DP, if the program is to be relocated, but we'll save that discussion for another chapter.)

There are two special "operators" that can also be used to affect the direct addressing mode, the less than sign (<) and greater than sign (>).

The less than sign can be used in an operand such as:

```
LDA    <$01
```

to force the assembler to assemble the instruction as a direct addressing type.

The greater than sign can be used for the opposite condition, to force extended addressing:

```
LDA    >$3001
```

In the latter case, even though the DP register holds $30, the LDA $3001 will assemble with "extended addressing."

Hints and Kinks 4-3
More on Pages

The 6800 predecessor to the 6809 used locations $00 through $FF, the first 256 bytes of RAM as "page 0." There was no DP register, and "direct addressing" could only be used for page 0. The 6809 adds the DP so that any of 256 pages can be used, by loading DP with $00 through $FF. One of the games on upgrading any microprocessor is to try to keep the new microprocessor "downwards compatible" with the previous version, to keep the industrial customers happy. 6809 instructions are not compatible with the 6800 on a "machine-language code" level, but the 6800 source code will run (more or less) on some 6809 assemblers.

When Should You Use Direct and When Extended Addressing?

You could go along very nicely and not have to worry about direct versus extended addressing. About the only effect you would see was when you tried to use operands from the first 256 locations — $00 through $FF. These instructions would always assemble as direct addressing types, as the DP register would be set to $00, and the assembler in EDTASM+ would automatically **impose** direct addressing. They wouldn't hurt your program one bit.

If you want to flex your programming muscles, however, you might try setting the DP register to a "page" that you access frequently, one which holds many "variables" or "constants," along with using the SETDP to inform EDTASM+ to which page DP was set. Any instruction that referenced those 256 locations would then be assembled as a direct addressing type.

If this concept bothers you, just ignore it for the time being and come back to it at a later time. It won't hurt your programs!

Hints and Kinks 4-4
ZBUG < and > Signs

You may have noticed the less than and greater than signs in ZBUG examination of memory locations. In the Mnemonic mode (#M), where the instruction mnemonic is displayed, you'll see a less than if the instruction uses direct addressing and a greater than if the instruction uses extended addressing. An example: Assembling

```
START     LDA      $0
          LDB      $3000
```

and then going to ZBUG to examine in Mnemonic mode displays

```
*Z
#START/ LDA <0          (indicates direct addressing)
 START+2/          LDB >3000   (indicates extended addr)
```

Review

To recap the material in this chapter:

- Extended addressing allows the programmer to address any of the 65,536 memory addresses available for memory; the address is held in the instruction itself as a 2-byte number
- Extended addressing uses an 8- or 16-bit operand from memory to perform the instruction operation
- If a 16-bit operand is used, the extended memory address points to the first byte of the operand
- Loads and stores transfer 8 or 16 bits of data between a cpu register and memory

- Simple indexed addressing uses the X or Y index register as a "pointer" register to point to data in memory
- The format of simple indexing is ",X" or ",Y" used as the operand of the instruction
- Direct addressing uses the contents of the DP register as the most significant 8 bits of the memory address; from that point on, it is identical to extended addressing
- The direct addressing mode saves one byte over the comparable extended addressing type and is also faster
- The EDTASM+ assembler automatically imposes direct addressing if an instruction references an operand in the current "Direct Page"
- The assembler must be informed of the current DP contents by a SETDP "pseudo-op" with an operand value of the DP contents
- The < and > operators are used to force direct and extended addressing, respectively
- It's not strictly necessary to worry about setting the DP, as it is set to $00 on power up or reset

For Further Study

TFR instruction for loading DP (see Appendix II)
SETDP pseudo-op (EDTASM+ manual)

KEY CHART — CHAPTER 5

INSTRUCTIONS

ABX	LBHS	~~CLRB~~	~~LDS~~	ROL
ADCA	BITA	CLR	~~LDU~~	RORA
ADCB	BITB	CMPA	~~LDX~~	RORB
ADDA	BLE	CMPB	~~LDY~~	ROR
ADDB	LBLE	CMPD	LEAS	RTI
ADDD	BLO	CMPS	LEAU	RTS
ANDA	LBLO	CMPU	LEAX	SBCA
ANDB	BLS	CMPX	LEAY	SBCB
ANDCC	LBLS	CMPY	LSLA	SEX
ASLA	BLT	COMA	LSLB	STA
ASLB	LBLT	COMB	LSL	STB
ASL	BMI	COM	LSRA	STD
ASRA	LBMI	CWAI	LSRB	STS
ASRB	BNE	DAA	LSR	STU
ASR	LBNE	DECA	MUL	STX
BCC	BPL	DECB	NEGA	STY
LBCC	LBPL	DEC	NEGB	SUBA
BCS	BRA	EORA	NEG	SUBB
LBCS	LBRA	EORB	NOP	SUBD
BEQ	BRN	~~EXG~~	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	JMP	PSHU	TFR
LBGT	LBVC	JSR	PULS	TSTA
BHI	BVS	~~LDA~~	PULU	TSTB
LBHI	LBVS	~~LDB~~	ROLA	TST
BHS	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	**T DISPLAY BLOCK**
~~B(YTE) MODE~~	**T H HARDCOPY BLOCK**
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	, SINGLE STEP

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
ADDITION AND SUBTRACTION	MULTIPLES
CONDITION CODES	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 5
Addition and Subtraction

In this lesson we'll discuss actual arithmetic in the 6809 — additions and subtractions. These basic operations are the cornerstone of 6809 assembly-language programming, and we'll be discussing how we can do rudimentary adds and subtracts. We'll also look at the ancient Egyptian art of using "two's complement" numbers. (Well, ok, maybe it doesn't go back 4000 years, but I know for certain its been in use since February.)

Eight-Bit Adds

The 6809 has one basic add: An 8-bit operand from memory is added to the contents of the A or B accumulators. The result is then put back into the A or B accumulator. Let's see how this works. Look at the source code below, or use the existing source code from the Lesson File:

```
                     00100 *8-BIT ADDS
090D 86     64       00110 START   LDA     #100     100 TO A
090F B7     3000     00120         STA     $3000    STORE IN RAM $3000
0912 C6     96       00130         LDB     #150     150 TO B
0914 FB     3000     00140         ADDB    $3000    ADD ($3000) AND B
0917 F7     3000     00150         STB     $3000    SAVE RESULT
091A 86     25       00160         LDA     #37      37 TO A
091C 8B     25       00170         ADDA    #37      ADD 37 TO A
091E B7     3001     00180         STA     $3001    SAVE RESULT
0921 7E     0921     00190 LOOP    JMP     LOOP     LOOP HERE
            0000     00200         END
00000 TOTAL ERRORS

LOOP    0921
START   090D
```

Figure 5-1. Eight-Bit Adds Program

Try this sequence yourself. Enter the code above except for the optional comments. Assemble the code to get an error-free assembly by A/IM.

Go to ZBUG, breakpoint at "LOOP," and execute from "START."

Here's what should happen on execution: The first LDA loads the A register with 100. This value is then stored in RAM location $3000. Next, the B register is loaded with 150. Now an ADDB is done with an operand of $3000. This ADDB will add the contents of the B register with the **contents** of location $3000. Since $3000 contained 100, we're adding 100 to 150 in B. The result of the add is now put into the B register. The STB then stores the result into location $3000.

The second portion of the program works like this: The A register is first loaded with 37 decimal. An ADDA now loads the **immediate** value of 37 to the A register. The result of 37+37 is put back into the A register. Finally, the result is stored into location $3001.

When you reach the breakpoint at LOOP, do a

    #R

to see the contents of the registers. You should see a hex $FA in B, which is 250, and a hex $4A in A, which is 74. Now examine RAM locations $3000 and $3001 by a slash examination. You should see a $FA in $3000 and a $4A in $3001.

Hints and Kinks 5-1
The T Examination Mode

The T examination mode in ZBUG can be used in place of examining consecutive locations by slash and DOWN ARROW. To use T in the above example, enter

```
#T3000 3001
```

The locations from $3000 through $3001 will be displayed. T will display any block of locations you specify. The bad news is that each will be displayed on one line of the screen, so you'll only be able to see the last 16 at any time unless you're an **extremely** fast reader.

The TH command works just like T except that it prints the locations on the system line printer.

```
TH3000 3100
```

will print the 257 locations from $3000 through $3100 on the printer.

In the program above, you can see two of the types of addressing modes that we can use with 8-bit adds.

The first is extended (the ADDB $3000). We could also use direct addressing if we had set up the DP register properly. Look at the object code to see the direct address; you'll see a $FB opcode byte followed by $30, followed by $00.

The second addressing mode used with A is immediate — the ADDA #37, for example. You can see the operand for the add in the ADDA #37 instruction. You'll see a $8B opcode byte followed by a $25 in the second byte.

These addressing modes — direct page, extended, and immediate, are typical for all instructions that use the A register. The same addressing modes would be available for a subtract, an OR operation, or an increment of A. As each addressing mode counts for a separate instruction type, you can see where some of the many separate instructions of the 6809 come from. Many instructions are just the same instruction with a different addressing mode!

Sixteen-Bit Adds

All 8-bit adds use the A or B registers. What would you expect 16-bit adds to use? Correct, the D register, which is a "16-bit" accumulator made up from A (most significant, or left half) and B (least significant or right half).

To see how the D register is used to add two 16-bit numbers, look at the source lines below:

```
                         00100 * 16-BIT ADDS USING D
0959 CC      03E8        00110 NEXT     LDD      #1000      1000 DECIMAL
095C FD      3000        00120          STD      $3000      STORE
095F CC      00FA        00130          LDD      #250       250 DECIMAL
0962 FD      3002        00140          STD      $3002      STORE
0965 CC      03E8        00150          LDD      #1000      1000 DECIMAL
0968 F3      3000        00160          ADDD     $3000      ADD ($3000) AND D
096B FD      3000        00170          STD      $3000      STORE BACK IN $3000
096E FC      3002        00180          LDD      $3002      LOAD ($3002)
0971 C3      0064        00190          ADDD     #100       ADD 100 DECIMAL
0974 FD      3002        00200          STD      $3002      STORE BACK IN $3002
0977 7E      0977        00210 LOOP     JMP      LOOP       LOOP HERE
             0000        00220          END
00000 TOTAL ERRORS

LOOP     0977
NEXT     0959
```

Figure 5-2. Sixteen-Bit Adds Using D Program

Assemble as before, breakpoint at LOOP, execute from NEXT, and then look at the register contents and locations $3000 through $3003.

Here's what the program above will do: The D register was first loaded with decimal 1000. This loads the A register with $03 and the B register with $E8. The D register is then stored in RAM location $3000. The STD here stores A in location $3000 and B in $3001.

Next, the D register is loaded with decimal 250 (A is loaded with 0 and B with $FA). This value is then stored in RAM locations $3002 ($00) and $3003 ($FA).

Next, the D register is loaded with decimal 1000. An ADD then adds the contents of RAM location $3000 (1000 decimal) to the contents of the D register (1000 decimal). The result is put into D. The result is 2000, with $07 going to A and $D0 going to B. This result is then stored back into $3000.

The next LDD loads the D register with the contents of $3002 (250 decimal) and then does an ADDD #100. This is an add that uses immediate addressing to add decimal 100 to D. The result of 350 is then stored in D by the ADDD, with $01 going to A and $5E going to B. The result is also stored back into location $3002 and $3003.

Look at the registers and memory using ZBUG to verify that all of these actions occurred. There should be a $015E in D for the last add, and also in $3002, $3003. Locations $3000 and $3001 should hold $07D0.

Of course we went through a lot of work here just to give you some more experience in loads and stores along with the adds without accomplishing much. You're going to have to suffer through a few more trivial examples, and then we'll start doing some interesting things.

How did these 16-bit adds differ from the 8-bit adds? For one thing, of course, twice the "width" of data was added, which means that sums up to 65,535 can be handled, rather than only 0 through 255. For another thing, the 16-bit adds are "register hogs." The 16-bit add makes use of the D register, which is both A and B, which means that we can't have anything important in either register.

Hints and Kinks 5-2
Working With 16-Bit Numbers

The second part of Appendix IV shows a procedure for converting 16-bit numbers. It depends upon the fact that the **most significant byte** of a 16-bit number is really an 8-bit number times 256. Take the 16-bit number 1010111011000000. This number is really 10101110*256+11000000 (174*256+192) as each bit position is a power of 2 and the eighth bit position is 2 to the 8th or 256. You can do two converts, one for each byte of the number and still use the tables in Appendix IV as described in the procedure.

Subtracts

The 8-bit subtract is very much like an 8-bit add. It operates on data in the A or B registers. In the subtract, an 8-bit operand from memory is subtracted from the contents of the A or B register, with the result going into the A or B register — it's identical to the ADD except for the basic operation.

To show you how the subtract works, look at the following code.

```
                  * 8- AND 16-BIT SUBTRACTS
0937 86   64      THIRD   LDA    #100     100 DECIMAL
0939 C6   FA              LDB    #250     250 DECIMAL
093B 80   0A              SUBA   #10      10 DECIMAL
093D B7   3000            STA    $3000    STORE RESULT
0940 F0   3000            SUBB   $3000    SUBTRACT ($3000)
0943 F7   3001            STB    $3001    STORE RESULT
0946 CC   00FA            LDD    #250     250 DECIMAL
0949 83   03E8            SUBD   #1000    SUBTRACT
094C FD   3002            STD    $3002    STORE RESULT
094F 7E   094F    LOOP    JMP    LOOP     LOOP HERE
          0000            END
00000 TOTAL ERRORS

LOOP    094F
THIRD   0937
```

Figure 5-3. Eight and Sixteen Bit Subtracts Program

You can see immediately that the subtract can use either A, B, or D, and it can also use direct, extended, or immediate addressing.

The A register is first loaded with decimal 100 and the B register with decimal 250. Next, an SUBA #10 is done, which subtracts the immediate value of decimal 10 from A. The result of decimal 90 is stored into A by the SUBA and into location $3000 by the following STA.

Next, a SUBB with extended addressing is done. This subtracts the value of

90 in $3000 from the contents of B (250) and puts the result of 160 into B. An STB also stores it in RAM location $3001.

The third subtract, an SUBD, subtracts an immediate value of 1000 decimal from the contents of D, which is 250 decimal. The result of -750 goes back into D, and is also stored into locations $3002 and $3003.

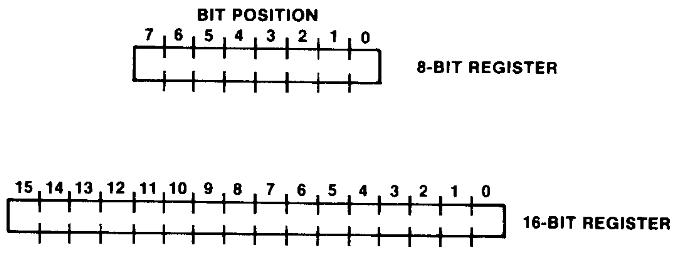
Use the R and T ZBUG commands to examine the registers and RAM locations $3000 through $3003 for the correct results.

If you're paying strict attention here, you've noticed a confusing situation. What about that last subtract? What is a negative value of -750? The result looked **strange.**

To answer that question we're going to have to look at number representation called "two's complement."

Hints and Kinks 5-3
Bit Numbering

Bit positions in both cpu registers and memory locations are numbered from right to left starting with 0, as shown below. The bit number corresponds to the power of 2 for that bit position. The leftmost bit position number in an 8-bit register is 7, for 2 to the 7th, while the leftmost bit position number in a 16-bit register is 15, for 2 to the 15th.



Two's Complement Numbers

Up to this time we've been working with "absolute numbers" held in 8 to 16 bits. The binary numbers we've considered have always been positive, integer values. This makes sense in many cases. Take X register values, for example. The X register is primarily meant to hold memory address values, 0 through 65,535, and there is no such thing as a "negative address."

However, we would like to be able to represent both positive **and** negative numbers. (I need **some** way to handle balances in my checking account. . .) How is it done?

The scheme that the 6809 and almost all other microprocessors or computers use is called "two's complement." The format of two's complement numbers is shown in Figure 5-4. Appendix V also lists all 8-bit two's complement numbers.

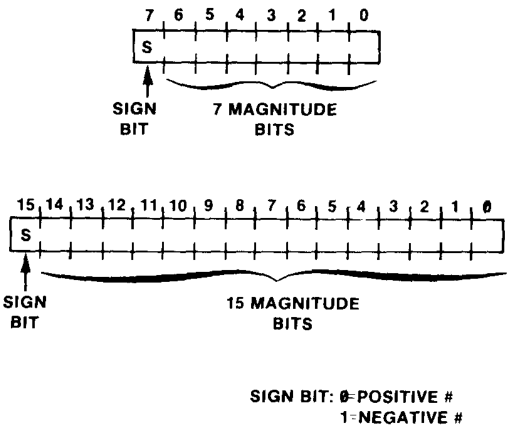


Figure 5-4. Two's Complement Format

The most significant bit of an 8 or 16-bit number is designated as a "sign bit." The remaining bits are "magnitude bits."

If the sign bit is a 0, well and good. The remaining 7 or 15 bits represent the "magnitude" of a positive number. In 8-bit values you can therefore have positive numbers from 0 0000000 (0) through 0 1111111 (+127).

If the sign bit is a 1, however, the number represents a negative value. In that case, change all the 0s to 1s, change all the 1s to 0s, and add 1. Why? A purely mechanical process that gives you the magnitude of the negative value.

Let's take a subtract and show you how it's done. See Figure 5-5 where we subtract 123 from 100. The result was $E9, or binary 11101001. The most significant bit was a 1, so the number is a negative number. Changing all 1s to 0s and all 0s to 1s gives us 00010110. Adding 1 gives us 00010111. 00010111 is 23 decimal, and therefore the negative number is -23, what we should expect to get by subtracting 123 from 100.

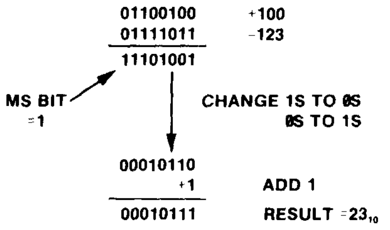


Figure 5-5. Two's Complement Example

Negative values of -1 (11111111) through -128 (10000000) can be held in 8 bits. (Note that although we can hold a negative number of -128, the maximum positive number that can be held is +127.)

Two's complement works exactly the same in 16 bits. The same actions are taken to convert. Look at the sign bit first, and if a 0, the number is a positive number from 0 (0000000000000000) through +32,767 (0111111111111111). If the sign bit is a 1, the number is a negative number from -1 (1111111111111111) through -32,768 (1000000000000000).

Note that the two's complement number ranges are the same as BASIC integer values. All BASIC integer values are in reality two's complement 16-bit values!

Why are two's complement numbers used? So that we can easily do adds and subtracts with the ADD and SUB instructions in the 6809. We don't have to laboriously check each number to see if it's positive or negative, we just go ahead and do the add or subtract — the number will be adjusted accordingly. We'll discuss this more in Chapter 10, along with other arithmetic operations.

Review

To review what we've learned in this chapter:

- Eight-bit adds use either the A or B registers and an 8-bit memory operand for the add
- Eight-bit adds can use direct page, extended, immediate, or other addressing modes
- Sixteen-bit adds use the D register as a "16-bit accumulator"
- Sixteen-bit adds add the operand from two memory locations to the contents of D
- Eight-bit and 16-bit subtracts are virtually identical to 8-bit adds and subtracts as far as handling of operands and addressing modes
- Two's complement numbers express both positive and negative integer values; if the sign bit is a 0, then the remainder of the number determines the positive magnitude, otherwise a simple conversion must be done to find the negative number

For Further Study

Two's complement numbers - Appendix V

KEY CHART — CHAPTER 6

INSTRUCTIONS

ABX	LBHS	~~CLRB~~	~~LDS~~	ROL
ADCA	BITA	CLR	~~LDU~~	RORA
ADCB	BITB	**CMPA**	~~LDX~~	RORB
ADDA	BLE	**CMPB**	~~LDY~~	ROR
ADDB	LBLE	**CMPD**	LEAS	RTI
ADDD	BLO	**CMPS**	LEAU	RTS
ANDA	LBLO	**CMPU**	LEAX	SBCA
ANDB	BLS	**CMPX**	LEAY	SBCB
ANDCC	LBLS	**CMPY**	LSLA	SEX
ASLA	BLT	COMA	LSLB	**STA**
ASLB	LBLT	COMB	LSL	**STB**
ASL	BMI	COM	LSRA	**STD**
ASRA	LBMI	CWAI	LSRB	**STS**
ASRB	BNE	DAA	LSR	**STU**
ASR	LBNE	DECA	MUL	**STX**
BCC	BPL	DECB	NEGA	**STY**
LBCC	LBPL	DEC	NEGB	**SUBA**
BCS	BRA	EORA	NEG	**SUBB**
LBCS	LBRA	EORB	NOP	**SUBD**
BEQ	BRN	~~EXG~~	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	JMP	PSHU	**TFR**
LBGT	LBVC	JSR	PULS	**TSTA**
BHI	BVS	~~LDA~~	PULU	**TSTB**
LBHI	LBVS	~~LDB~~	ROLA	**TST**
BHS	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	**, SINGLE STEP**

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
CONDITION CODES	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 6
Using the Condition Codes

The Condition Codes, also called "CC," are a collection of 8 bits. In this chapter we're going to investigate the Condition Codes of the 6809. The Condition Codes record the results of comparisons between operands and are indispensable to conditionally altering the program flow with an assembly-language program.

The 8 Condition Codes (Figure 6-1) reflect the results of arithmetic and other instructions, as we'll see. The Condition Codes can alter the program flow, as they can be tested by "branch" instructions, which cause jumps to different program code on the settings of the Condition Codes, which in turn are set by prior arithmetic.

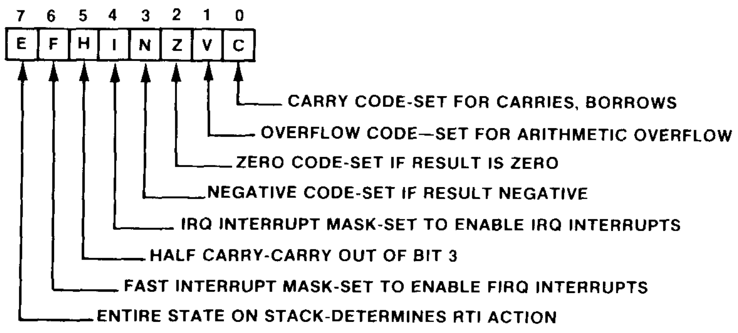

7 6 5 4 3 2 1 0
E F H I N Z V C
CARRY CODE-SET FOR CARRIES, BORROWS
OVERFLOW CODE—SET FOR ARITHMETIC OVERFLOW
ZERO CODE-SET IF RESULT IS ZERO
NEGATIVE CODE-SET IF RESULT NEGATIVE
IRQ INTERRUPT MASK-SET TO ENABLE IRQ INTERRUPTS
HALF CARRY-CARRY OUT OF BIT 3
FAST INTERRUPT MASK-SET TO ENABLE FIRQ INTERRUPTS
ENTIRE STATE ON STACK-DETERMINES RTI ACTION

Figure 6-1. 6809 Condition Codes

Here's an example:

```
SUBA  $3000     A — ($3000)
BEQ   EQUAL     GO IF A=($3000)
                NOT EQUAL HERE
```

The first instruction above subtracted the contents of the byte at RAM location $3000 from the A register. The subtract instruction affects most Condition Codes. If the result is zero, the Z Condition Code is set as one of the last actions in the subtract. The BEQ instruction is a "conditional jump" that jumps to location EQUAL if the Z Condition Code is set, but otherwise doesn't "take the jump" and "falls through" to the next instruction in sequence. The mnemonic "BEQ" means "Branch if Equal"; in this case, the BEQ actually tests the Z Condition Code to see if it is set because of the equality of A and the contents of location $3000.

Set means a 1 condition, that is, Z=1.

More on Adds and Subtracts — the Z Condition Code

Let's look at some of the Condition Code actions during 8-bit adds and subtracts, as we're already experts on these! Enter the code below.

```
                      00100 * ARITHMETIC CC ACTIONS
08CB 86    21         00110 START    LDA     #33      A OPERAND
08CD 80    21         00120          SUBA    #33      33-33 0
08CF C6    20         00130          LDB     #32      B OPERAND
08D1 C0    21         00140          SUBB    #33      32-33
08D3 86    22         00150          LDA     #34      A OPERAND
08D5 80    21         00160          SUBA    #33      34-33
08D7 7E    08D7       00170 LOOP     JMP     LOOP     LOOP HERE
           0000       00180          END
00000 TOTAL ERRORS

LOOP     08D7
START    08CB
```

Figure 6-2. Arithmetic Flag Actions Program 1

You can easily see the results of these operations using ZBUG. Assemble the code so that you get an error-free assembly by A/IM.

```
Go to ZBUG. Now enter
START,
```

that's the label START followed by comma. This ZBUG instruction will allow you to "single step" through any program an instruction at a time. After you enter the START, ZBUG will immediately execute the instruction at start and you'll see

```
#START,
 START+2/ SUBA #21
```

At this point you can do an R command to display the registers. Note the CC= display. The CC= will be followed by a hex code, which represents the Condition Codes as shown in Figure 6-1. To the right of the hex code, you'll see another equals, followed by mnemonics of whatever Condition Codes are set (equal to 1). The mnemonics correspond to the mnemonics shown in the figure.

After the first "single-step," do another by entering only a comma

```
#Z
#START,
 START+2/ SUBA #21
#R
 A=XX  B=XX  DP=XX  CC=XX =
 X=XXXX  Y=XXXX  U=XXXX  S=XXXX
 PC=XXXX
#,
```

Use a comma to single step through the six instructions. After each single step, do another R to observe the registers.

Hinks and Kinks 6-1
More on Single Step

The single step command in ZBUG (a comma) is very powerful because you can step your way through a program an instruction at a time. In between instruction steps, you can use other ZBUG commands to look at registers (R), memory (slash or T), or get hardcopy (TH). Don't worry about affecting the contents of registers or memory by the other commands, the 6809 register contents will remain intact from step to step.

When a single step comma command is executed, it displays the **next instruction to be executed,** so you'll have to look at the preceding instruction from a listing or from memory.

Here's what you'll see: The first subtract subtracted 33 from 33. The result of this was 0 in the A register, so the Z Condition Code should have been set (1) to indicate a Zero condition. You'll see a Z after the second equals. The next subtract subtracted 33 from 32. The result of this is -1, or FFH, definitely a "non-zero" condition, and the Z Condition Code should have been reset (0) to indicate non-zero. You'll see no Z after the equals. The third subtract subtracted 33 from 34; this is also a non-zero result, and the Z Condition Code should have remained reset at 0 for no Z.

If you didn't catch the Condition Codes the first time, run the program again and try to observe the Z Condition Code.

The Z Condition Code, therefore, is set or reset according to the results of the subtract. It is also normally affected by other instructions although certain instructions, such as branches, leave the Z and other Condition Codes unchanged.

In most cases you'll be using the Condition Code setting directly after the add, subtract, or other processing in a conditional jump, so there won't be intervening instructions that could affect the Condition Codes. In other cases the "test" of the Condition Codes by a conditional jump may be several instructions away. It's important, therefore, to know which instructions affect the Condition Codes, and which ones do not. You can find this information in Appendix II, where all Condition Codes are listed for every instruction type.

The N(egative) Condition Code

Another Condition Code in the CC register is the N Condition Code, standing for "Negative." The N Condition Code is set (1) if the result of the operation is negative, and reset (0) if the result of the operation is positive. Here again, the N Condition Code is usually affected, but check Appendix II to make certain. N is affected by adds, subtracts, and other arithmetic and logical operations.

Single step the program above again, but this time watch the settings of the N Condition Code.

You should have seen the N reset (0) as the first subtract of 33-33 was done.

This indicates that the result is a positive number. (Zero is always a positive number in the 6809. If we apply the rules of two's complement "notation" and look at the sign bit, we see a positive number with a magnitude of 0000000.)

The next subtract was 32-33. The result here should have been a negative value of -1. The N bit here was set (1) after the subtract to indicate that the result was negative, and you should have seen an N mnemonic after the second equals for the CC.

The last subtract was 34-33, for a result of 1, a positive number again. The N Condition Code was reset (0) after this subtract.

Here's one of those interesting questions (that usually occur at 2:00 a.m.) How does the 6809 know whether we are subtracting in two's complement, or in absolute form? In other words, we might be working with absolute numbers in A from 00000000 through 11111111 (255), instead of two's complement numbers of -128 through +127? The answer is: The 6809 **doesn't** know! It blithefully sets the N Condition Code as if it were two's complement operations. However, we know, and if we are operating in absolute numbers up to 255, we ignore the N Condition Code.

The Compare

There's an important variation of the SUB instruction that we've ignored up to this point. This is the CMP, or Compare instruction. The compare works exactly like the SUB, except that it does not put the result back into the A register. It simply drops the result into the "bit bucket" on the floor behind the Color Computer.

What the compare **does** do, however, is to set the Z, N, and other Condition Codes. We can use the compare to test one operand against another without destroying the contents of the A register.

```
                         00100 * ARITHMETIC CC ACTIONS
08DA 86    21            00110 START1   LDA      #33      A OPERAND
08DC 81    21            00120          CMPA     #33      33-33=0
08DE CC    1000          00130          LDD      #$1000   A OPERAND
08E1 FD    3000          00140          STD      $3000    STORE
08E4 8E    1001          00150          LDX      #$1001   OPERAND
08E7 BC    3000          00160          CMPX     $3000    $1001-$1000
08EA 7E    08EA          00170 LOOP     JMP      LOOP     LOOP HERE
           0000          00180          END
00000 TOTAL ERRORS

LOOP     08EA
START1   08DA
```

Figure 6-3. Arithmetic Flag Actions Program 2

If you single step this program you can see how the Z Condition Code and N Condition Code change for the CMPs. Also note that the A and X registers don't change when the CMP is executed. Note that the CMP is different from the subtract because it can be used with X. In fact, you can do a compare for A, B, D, S, U, X, or Y (CMPA, CMPB, CMPD, CMPS, CMPU, CMPX, or CMPY).

The CMP instruction is probably used more frequently than the SUB and that's probably why it can be used with all registers. You can use direct page, extended, immediate, or simple indexed addressing with the CMP, along with addressing modes that we haven't covered, such as more involved indexing.

A Special CMP

There's another instruction that is very similar to the CMP. It's the TST instruction which may be used with A or B (TSTA, TSTB) or with an operand in a memory location (TST). The TST **only** sets the N and Z Condition Codes based on the contents of A, B, or the specified memory location. It's used for a "quick" test of an operand without having to waste an instruction or time in doing an actual comparison to data.

```
        TSTA                      TEST CONTENTS
                                  OF A (N AND Z)
        TST            $3000      TEST ($3000)
```

The Overflow (V) Condition Code

The Overflow Condition Code, abbreviated V, is used to record an **overflow** condition for arithmetic instructions like the ADD, SUB, and CMP. Overflow occurs when the result of an add or subtract is too large to be held in 8 or 16 bits. Examples are an add of 120 and 20 in 8 bits, or a subtract of -30,000 from +10,000 in 16 bits. Both results will set the V Condition Code to 1, indicating an overflow condition.

For an example of the V Condition Code operation, see the following code:

```
* V FLAG OPERATION
ANUDR   LDA     #100      100
        ADDA    #100      100+100=200=V!
        LDD     #-30000   -30000
        ADDD    #-30000   -60000
LOOP    JMP     LOOP      LOOP HERE
        END
```

Single step through this code and keep a sharp eye on the V Condition Code mnemonic when doing an "R" register display.

What did you see? The first ADD added an immediate 100 to the 100 in the A register for a result of 200. This is an overflow condition, and the V Condition Code should have been set to a 0, indicating no overflow; you should not have seen a "V."

The next add added an immediate -30000 to the -30000 in the D register to itself. This is an overflow condition and you'd see that the V Condition Code was set to indicate overflow for this instruction!

Hints and Kinks 6-2
Overflow on Absolute Numbers

What about overflow on absolute numbers? The Overflow Condition Code is not valid for arithmetic operations on absolute (unsigned) numbers, but there are certainly overflow conditions, as in adding $3000 to $E000 to compute a memory address (the result of $11000 cannot be held in 16 bits). Use the Carry Condition Code to check that the result did not exceed 255 or 65535 on an add; it will be set if overflow occurred. On a subtract use the conditional Branch "BLO" to detect if the result "went negative" (conditional branches are discussed in the next two chapters).

The C(arry) Condition Code

The Carry Condition Code is used in many different operations in the 6809. Its original use was to hold the state of the carry from the most significant bit of an add, or a borrow to the next bit on a subtract, as shown in Figure 6-4.

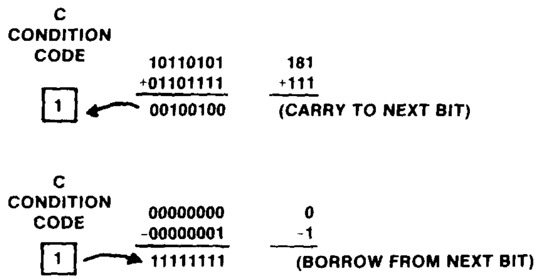


Figure 6-4. Carry Condition Code Actions

Another use of the Carry Condition Code is to hold the state (0 or 1) of the most significant bit on a "shift" or "rotate" operation. We'll look at these applications of the Carry Condition Code in future chapters.

Review

To review what we've learned in this chapter:

- There are 8 Condition Codes in the CC register; they are grouped together as the CC register
- The Z Condition Code is set after arithmetic and other instructions when the result is zero; it is reset when the result is non-zero

- The N Condition Code is set after arithmetic and other instructions when the result is negative; it is reset when the result is positive
- The 6809 acts as if two's complement numbers were being processed in setting the N Condition Code, but the arithmetic may be "absolute"
- The Compare instruction CMP acts like a Subtract in Condition Code settings, but does not put the result into the register
- The V Condition Code is set to indicate overflow conditions for both 8 and 16-bit operations
- The Carry Condition Code is used to record the carry or borrow from a high-order bit or the state of a bit on a shift or rotate
- The Condition Codes are affected for most instructions, but not affected by others; the programmer must be aware of when they are affected

For Further Study

Appendix II: CC Settings

KEY CHART — CHAPTER 7

INSTRUCTIONS

ABX	LBHS	~~CLRB~~	~~LDS~~	ROL
ADCA	BITA	CLR	~~LDU~~	RORA
ADCB	BITB	~~CMPA~~	~~LDX~~	RORB
~~ADDA~~	BLE	~~CMPB~~	*LDY*	ROR
~~ADDB~~	LBLE	~~CMPD~~	LEAS	RTI
~~ADDD~~	BLO	~~CMPS~~	LEAU	RTS
ANDA	LBLO	~~CMPU~~	LEAX	SBCA
ANDB	BLS	~~CMPX~~	LEAY	SBCB
ANDCC	LBLS	~~CMPY~~	LSLA	SEX
ASLA	BLT	COMA	LSLB	~~STA~~
ASLB	LBLT	COMB	LSL	~~STB~~
ASL	BMI	COM	LSRA	~~STD~~
ASRA	LBMI	CWAI	LSRB	~~STS~~
ASRB	**BNE**	DAA	LSR	~~STU~~
ASR	LBNE	DECA	MUL	~~STX~~
BCC	**BPL**	DECB	NEGA	~~STY~~
LBCC	LBPL	DEC	NEGB	~~SUBA~~
BCS	**BRA**	EORA	NEG	~~SUBB~~
LBCS	LBRA	EORB	NOP	~~SUBD~~
BEQ	BRN	~~EXG~~	ORA	SWI
LBEQ	LBRN	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	BVC	**JMP**	PSHU	*TFR*
LBGT	LBVC	JSR	PULS	*TSTA*
BHI	BVS	~~LDA~~	PULU	*TSTB*
LBHI	LBVS	~~LDB~~	ROLA	*TST*
BHS	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABL~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	← BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC.BYT
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
SYMBOLIC ADDRESSING	DECIMAL ARITHMETIC
JUMPS, BRANCHES	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type = Present Chapter
Regular Type = Future Chapters
~~Italic Type~~ = *Past Chapters*

Chapter 7
Symbolic Addressing, Jumps, and Branches

Assembly language uses symbols to represent absolute memory locations. These symbols can be referenced in jumps and branches, instructions that are similar to BASIC "GOTO"s or "IF. . .THEN" commands. In this chapter we'll see how the 6809 performs "loops" or "iterative" operations by means of branching instructions.

A loop is simply two or more cycles through the same set of instructions. For an example of a simple loop, see the code below:

```
                         * SIMPLE LOOP
08B4 8E     0064         SIMLOP  LDX     #100     LOAD X WITH 100
08B7 BF     3000                 STX     $3000    STORE 100 IN $3000
08BA CC     0000                 LDD     #0       CLEAR D
08BD F3     3000         SIM010  ADDD    $3000    ADD 100 TO D
08C0 7E     08BD                 JMP     SIM010   LOOP
            0000                 END
00000 TOTAL ERRORS

SIM010   08BD
SIMLOP   08B4
```

Figure 7-1. Simple Loop Program

If you've assembled this code, you can single step in ZBUG by using "SIMLOP," followed by a single comma for each instruction step.

If you use the R command and single step as the program executes, you can see that the D register is incremented by 100 each time through the loop. Notice also how the program counter display changes between the location of SIM010 and the instruction following.

The JMP SIM010 instruction is an "unconditional" jump instruction. It **always** jumps back to a specified address.

Look at the three machine-language bytes for the JMP SIM010 instruction. The first is an "opcode" byte of $7E. The next two are address bytes for the instruction.

The value of the two address bytes correspond to the location of the label SIM010. You can verify this by looking at the assembly listing or by the "symbol table" values printed out at the end of the assembly.

Symbolic Addressing

Using a label instead of an absolute address in memory is termed "symbolic addressing." It relieves the programmer of having to compute the actual address for the Jump. Of course, you could still "hand assemble" machine-language object code, but it's much more convenient to let the assembler do it for you.

We've used labels for breakpointing and execution in ZBUG in previous chapters, but this is the first time that we're using them for their primary purpose — "tagging" an instruction for a Jump point. (Other than for the LOOP JMP LOOP we've included as "protection" for preceding programs.)

EDTASM+ and all assemblers build a table of symbols (appropriately enough called a "symbol table"). In it are all labels and other symbols encountered in the program. The assembler uses the symbols to "build" the addresses in instructions, as, for example, the address of SIM010 in the program above. The symbol table is dumped at the end of an assembly, unless you use the special assembler "switch" /NS to avoid displaying or printing the symbol table.

You can use labels for locations anytime you wish. The labels are usually associated with jump points, but do not have to be. Labels may be 1 to 6 characters, the first of which starts with an alphabetic character. One convention that I've used here, and that other programmers often use, is to make the primary label a 6-character descriptive label, and to make following labels the first 3 characters of the "module," followed by 3 digits. Often the digits will be in order, as in BASIC lines. You might have SQROOT as the first label of a square root code segment, for example, and the labels following would be SQR010, SQR020, SQR080, and SQR090.

Labels can also be used for immediate values as in

```
LOCA    LDA    #20        LOAD A WITH 20
        LDX    #LOCA      LOAD X WITH LOCA
```

which loads X with the **address** of the instruction at LOCA.

Hints and Kinks 7-1
Nested Loops

We've indented the comment portions of source code lines to indicate loops in this book. The further to the right the comment is indented, the "lower level" the loop is. Loops can be nested to any level, but typically you won't have more than about two or three levels of loops.

One of the nicest features of ZBUG is its ability to access the assembler symbol table and find out where the labels in the program are. (This feature is known as "symbolic debugging.") That's how you can refer to a symbolic location such as "LOOP" instead of using an absolute location in breakpointing.

Unconditional Jumps

The JMP instruction above is one of the unconditional jumps in the 6809 (there are other "BRanch" jumps that we'll talk about in the next chapter). It always has the same format of a $7E opcode, followed by two address bytes, and always transfers control to the jump address. Note that the jump address can be anywhere in memory — location 0 through 65,535; of course some of these addresses are invalid in the Color Computer as they define ROM or I/O addresses (see Figure 2-1).

The symbolic address used in the JMP is a common way of specifying the JMP point. Could you have used an absolute memory address such as $3056? Sure, but you wouldn't know what the values were until after you assembled. You'd have had to make up some dummy values, do a single assembly pass, find out the actual locations, and then fill in the proper addresses. With symbolic addressing, the assembler does it for you.

Hints and Kinks 7-2
Using Labels as Immediate Values

What happens when labels are used as immediate values, as in the example above? The assembler sees that an immediate value is to be loaded by the "#" sign prefix. It then checks to see if the operand is numeric. If it isn't, it then assumes that it is a symbol. The symbol table is then scanned to see if there is a symbol "match." As the symbol table holds all labels and their corresponding address values, the symbol (label) should be found, and it should represent an absolute location value. This value is then used in the immediate load just as if a numeric value was specified in the first place. Typically, the X or Y registers might be loaded with the starting address of a table or block of variables in this fashion. The start of the table might be called "TABLE" and an LDY #TABLE would load the TABLE location into Y.

Hints and Kinks 7-3
The BRA Instruction

The Branch Always instruction is very similar to a JMP instruction except that it uses 2 bytes instead of 3. The JMP can jump anywhere in memory and normally uses direct or extended addressing. The BRA uses a form of addressing called "relative addressing" that we'll look at in the next chapter.

Originally Motorola engineers had even more Branch instructions, but at the last minute, such useful branches as "Branch if Tuesday (BTU)" and "Branch Occasionally (BOC)" were left out of the 6809 design. Lucky for us...

Conditional Jumps (Branches)

The loop in the program above has one drawback; it never stops. We can easily control the loop, however, with a "conditional" jump. As an example of a loop with a conditional jump, look at this code:

```
                        00100 *LOOP WITH CONDITIONAL BRANCH
091F 4F                 00110 ADDNUM  CLRA                ZERO TOTAL
0920 C6     0A          00120         LDB     #10         COUNTER
0922 F7     3000        00130         STB     $3000       STORE
0925 BB     3000        00140 ADD010  ADDA    $3000         ADD 10+9+8+7...
0928 C0     01          00150         SUBB    #1            COUNT-1
092A F7     3000        00160         STB     $3000         SAVE COUNT
092D 26     F6          00170 LOPEND  BNE     ADD010        LOOP IF NOT 0
092F 7E     092F        00180 LOOP    JMP     LOOP        LOOP HERE
            0000        00190         END
00000 TOTAL ERRORS

ADD010   0925
ADDNUM   091F
LOOP     092F
LOPEND   092D
```

Figure 7-2. Loop With Conditional Branch

This short program adds the numbers from 1 to 10; at the end of the program, the total is in the A accumulator.

The "loop" from label ADD010 through label LOPEND is repeated 10 times. The first time, 10 is added to A (initially set to 0), the next, 9 is added, the next, 8, and so forth, down to 0.

Let's look at the code in detail. The CLRA instruction zeroes the total in A. A will be used to hold the runnng total.

The next instruction loads the B register with 10. B will be used to hold the current number and will start with 10 and "decrement" down to 0. The next instruction stores the current number into memory location $3000. This location will hold the current count also, as the B register cannot be added to A, but the second operand for the add must come from somewhere in memory.

The loop starts at ADD010. Each time through the loop, the following actions occur:

- The contents of RAM location $3000 is added to the contents of A with the result going into A.
- A SUBB #1 is done to subtract 1 from the count in B. B is initially 10. After the first SUB, it is 9, after the second, it is 8, and so forth. The count is then stored in $3000.
- The Z Condition Code is set on the result of the SUBB #1. If the Z flag is reset (Z=0), then A is not 0; if the Z flag is set (Z=1), then A is 0.
- The BNE instruction tests the Z flag. If it is not set (NE), the jump is made just as if the BNE was an unconditional JMP. If the Z flag is set, then the

BNE "falls through" to the end instruction and 10 passes through the loop have been made.

Conditions for Branches

We used an "NE" for the condition in the BNE. This is equivalent to "Branch if Not Equal" . Logically, this is the same as "Branch if Not Zero." The mnemonics for a conditional Branch may be somewhat confusing, so we'll list some of them here:

Mnemonic	Meaning	Flag Setting for BR
BEQ	Branch if Equal (Zero)	Z=1
BNE	Branch if Not Equal (non-Zero)	Z=0
BCS	Jump if Carry Set	C=1
BCC	Jump if Carry Clear	C=0
BMI	Jump if Minus	N=1
BPL	Jump if Positive	N=0

These are a few of the mnemonics for conditional branches. We'll be discussing others in later chapters.

Here are some examples of the use of these mnemonics in BR instructions:

```
SUBB      $3000       B-($3000)
BEQ       ZERO        JUMP IF ($3000)=B
BPL       PLUS        JUMP IF ($3000)<B
BMI       MINUS       JUMP IF ($3000)>B
```

In the first of these instructions, the contents of $3000 is subtracted from A.

The result is either 0, greater than zero, or negative. The BEQ tests the Zero Condition Code. If the Z Condition Code is set (EQ condition), then a branch is made to location ZERO.

If the BPL instruction is executed, then the contents of $3000 cannot equal B. A test is made of the Negative Condition Code by the mnemonic "PL" for plus. If the N Condition Code is reset (PL condition), then a branch is made to location PLUS.

If the BMI instruction is executed, then the contents of $3000 cannot equal B or be less than B. As a matter of fact, the result must be negative here, and the N Condition Code must be set (N). The BMI MINUS always results in a BR!

A Comparison Test Using Modify Memory

Let's expand the code above into a full-fledged comparison test of two memory locations:

```
                        ØØ1ØØ * COMPARISON TEST OF $3ØØØ AND $3ØØ1
Ø985 B6    3ØØØ         ØØ11Ø COMPTS   LDA      $3ØØØ    FIRST OPERAND
Ø988 BØ    3ØØ1         ØØ12Ø          SUBA     $3ØØ1    FIRST OP-SECOND OP
Ø98B B7    3ØØ2         ØØ13Ø          STA      $3ØØ2    STORE RESULT
Ø98E 26    Ø2           ØØ14Ø          BNE      NEXT1    GO IF NOT Ø
Ø99Ø 2Ø    Ø8           ØØ15Ø          BRA      STORE    WIND UP
Ø992 2B    Ø4           ØØ16Ø NEXT1    BMI      NEXT2    GO IF FIRST<SECOND
Ø994 86    Ø1           ØØ17Ø          LDA      #1       1 TO A
Ø996 2Ø    Ø2           ØØ18Ø          BRA      STORE    GO TO STORE
Ø998 86    FF           ØØ19Ø NEXT2    LDA      #-1      -1 TO A
Ø99A B7    3ØØ2         ØØ2ØØ STORE    STA      $3ØØ2    STORE RESULT
Ø99D 7E    Ø99D         ØØ21Ø LOOP     JMP      LOOP     LOOP HERE
           ØØØØ         ØØ22Ø          END
ØØØØØ TOTAL ERRORS

COMPTS   Ø985
LOOP     Ø99D
NEXT1    Ø992
NEXT2    Ø998
STORE    Ø99A
```

Figure 7-3. Comparison Test of Two Locations Program

The program compares two operands each from 0 to 127 at locations $3000 (operand 1) and $3001 (operand 2). The result of the comparison is put into location $3002. The result will be as follows:

Condition	Result ($3002)
(Op 1=Op 2)	0
(Op 1>Op 2)	1
(Op 1<Op 2)	-1

To run the program you must first put two operands you want to compare into locations $3000 and $3001. If you want to run the program you can do this in ZBUG by using the slash command to modify memory.

```
#3000/    1        (DOWN ARROW)
 3001/    2        (ENTER)
```

Breakpoint at LOOP by

```
#XLOOP
```

and execute by

```
#GCOMPTS
```

After the breakpoint, examine location $3002 by the slash command in ZBUG.

At the end of execution, you should see the contents of location $3002 changed to reflect the results of the comparison. It should be a 0, 1, or -1 (equals, greater than, or less than).

The program works similarly to the earlier version, and we'll leave it up to you to scrutinize it.

Review

To review what we've learned in this chapter:

- Loops are portions of code that are executed more than once in sequence
- Labels used in source lines generally give the line a "name" that the assembler will reference for jump addresses and which ZBUG can use
- Labels are 1 to 6 characters long and start with an alphabetic character
- There are several "unconditional" jumps in the 6809 that **always** jump to the jump address
- Labels can also be referenced for loading immediate data where the data is an address, such as a jump address
- "Conditional" branches jump if the condition is met, but otherwise do nothing
- Conditional branches test the state of the Zero Condition Code, the Negative Condition Code, the Carry Condition Code, and others
- Conditional branches use the following mnemonics: BEQ, BNE, BCS, BCC, BMI, BPL, and others

For Further Study

Appendix II — check the Condition Code actions for conditional and unconditional branches

KEY CHART — CHAPTER 8

INSTRUCTIONS

ABX	**LBHS**	~~CLRB~~	~~LDS~~	ROL
ADCA	BITA	CLR	~~LDU~~	RORA
ADCB	BITB	~~CMPA~~	~~LDX~~	RORB
~~ADDA~~	**BLE**	~~CMPB~~	~~LDY~~	ROR
~~ADDB~~	**LBLE**	~~CMPD~~	LEAS	RTI
~~ADDD~~	**BLO**	~~CMPS~~	LEAU	RTS
ANDA	**LBLO**	~~CMPU~~	LEAX	SBCA
ANDB	**BLS**	~~CMPX~~	LEAY	SBCB
ANDCC	**LBLS**	~~CMPY~~	LSLA	SEX
ASLA	**BLT**	COMA	LSLB	~~STA~~
ASLB	**LBLT**	COMB	LSL	~~STB~~
ASL	**BMI**	COM	LSRA	~~STD~~
ASRA	**LBMI**	CWAI	LSRB	~~STS~~
ASRB	~~BNE~~	DAA	LSR	~~STU~~
ASR	**LBNE**	DECA	MUL	~~STX~~
~~BCC~~	~~BPL~~	DECB	NEGA	~~STY~~
LBCC	**LBPL**	DEC	NEGB	~~SUBA~~
~~BCS~~	~~BRA~~	EORA	NEG	~~SUBB~~
LBCS	**LBRA**	EORB	**NOP**	~~SUBD~~
BEQ	**BRN**	~~EXG~~	ORA	SWI
LBEQ	**LBRN**	INCA	ORB	SWI2
BGE	BSR	INCB	ORCC	SWI3
LBGE	LBSR	INC	PSHS	SYNC
BGT	**BVC**	~~JMP~~	PSHU	~~TFR~~
LBGT	**LBVC**	JSR	PULS	~~TSTA~~
BHI	**BVS**	~~LDA~~	PULU	~~TSTB~~
LBHI	**LBVS**	~~LDB~~	ROLA	~~TST~~
BHS	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
RELATIVE
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYT
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
RELATIVE BRANCHES	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type Present Chapter
Regular Type Future Chapters
~~*Italic Type*~~ - *Past Chapters*

Chapter 8
Relative Branches, Conditional and Unconditional

"Relative" Branches are instructions that use the relative addressing mode to jump conditionally, based on the Condition Code settings, or unconditionally. The relative addressing mode is a special one-byte mode (plus one byte of opcode) that saves instruction bytes by making the branch address relative to the location of the Branch instruction itself. In this chapter we'll look at the format of Relative Branches, and the types of conditional Branches that can be made.

To see the basic difference between the two absolute jumps (such as JMP) and relative jumps (such as BEQ) let's look at a sample program that uses a BR type jump, or branch. (By the way, when we say "jump" or "branch" it means exactly the same thing.)

```
                     00100 * RELATIVE BRANCHES
0991 FC   3000       00110 RELJRS  LDD    $3000   LOAD SQUARE
0994 8E   FFFF       00120         LDX    #-1     CLEAR SQUARE ROOT
0997 108E 0001       00130         LDY    #1      INITIALIZE ODD INTEGER
099B 10BF 3002       00140         STY    $3002   STORE IN MEMORY VARIABLE
099F 30   01         00150 REL010  LEAX   1,X     SQUARE ROOT+1
09A1 31   3E         00160         LEAY   -2,Y    ODD INTEGER -2
09A3 10BF 3002       00170         STY    $3002   STORE ODD INTEGER
09A7 F3   3002       00180         ADDD   $3002   SQUARE-ODD INTEGER
09AA 25   F3         00190         BCS    REL010  LOOP IF NOT MINUS
09AC BF   3003       00200         STX    $3003   STORE SQUARE ROOT
09AF 7E   09AF       00210 LOOP    JMP    LOOP    LOOP HERE
          0000       00220         END
00000 TOTAL ERRORS

LOOP     09AF
REL010   099F
RELJRS   0991
```

Figure 8-1. Relative Branches Program

If you'd like to see how this program works, enter the source code above and assemble by A/IM.

The program above ties together a lot of the concepts that we have discussed in previous lessons into a program that will calculate square roots. As you know by now, the 6809 doesn't even have the capability to divide numbers (although it does have a Multiply instruction); developing a square root program is therefore not a minor accomplishment.

Let's see how the program works: A square root of a number is a number which when multiplied by itself will give the number, in case you're rusty. The square root of 100, for example, is 10, as 10 times 10 is 100. The square root of 169 is 13, as 13 times 13 is 169. The square root of 178 is 13.34.

One way to find a square root is to take the "square" and start subtracting "odd integers" - 1, 3, 5, 7, 9, and so forth from it. The number of subtracts that can be made is the square root. Don't ask me how it works, but it does!

Take a square of 102, for example. 102-1 is 101-3 is 98-5 is 93-7 is 86-9 is

77-11 is 66-13 is 53-15 is 38-17 is 21-19 is 2-21 is -19. We were able to subtract 1, 3, 5, 7, 9, 11, 13, 15, 17, and 19 from 102, a total of 10 odd integers, so the square root is 10. In this method we don't get the fractional part of the square root, only the "integer part."

To run the program, use the slash command of ZBUG to enter a square into locations $3000 and $3001. The number 1000, for example, would be $03, followed by $E8.

After entering the value and verifying that it is correct, run the program by breakpointing LOOP and starting at RELJRS.

```
*A/IM                   (assemble)
*Z                      (go to ZBUG)
#B                      (Byte mode)
#3000/        03        (DOWN ARROW)
#3001/        E8        (ENTER)
#XLOOP                  (breakpoint LOOP)
#GRELJRS                (start at RELJRS)
```

At the end of the program, you'll see the integer square root in locations $3003 and $3004; use the slash or T commands in ZBUG to examine the result. If you used a 1000 as the square, for example, you'll see 31 decimal or $1F in locations $3003 and $3004 taken together as 16 bits ($00 in $3003 and $1F in $3004).

Hints and Kinks 8-1
Square Root Program Notes

This is not the ultimate square root program, but it is easy to implement. The square root result is found to the next lowest integer. "Scale up" the square for more precision by multiplying by a "finagle factor" and then dividing by the square root of the finagle factor. To find the square root of numbers up to 655, for example, multiply the square by 100, perform the square root program, and then divide the result by 10. Example: Find square of 200. 200*100=20000. Square root of 20000 by program is 141. Divide by 10 to get answer of 14.1, one more digit of precision than 14, the answer without "scaling."

The square root is a 16-bit number, even though it cannot be more than 255. The most significant byte (in $3003) will always be 0, and the next byte (in $3004) will be the 8-bit result.

Let's review the steps of the program:

First, the D register was loaded with the square from locations $3000 and $3001. This was an "extended load" of two bytes.

Next, the X register was loaded with -1. The X register will have 1 added to it each time through the loop of the program. We must start off with -1 so that the first add of 1 results in 0.

Next, the Y register was loaded with +1. The Y register holds the "odd integer" of 1, 3, 5, and so forth. Y will have 2 subtracted from it each time through the loop. When this is done the first time, the initial +1 value becomes a -1. The Y register is then stored into location $3002. The odd integer in $3002 will be added to the square in D each time through the loop. An add of a negative value is the same as a subtract of a positive number, after all.

The loop starts at REL010. You can see the indented comments that indicate the instructions that are part of the loop.

Each time through the loop, these things happen:

One is added to the X register. This "bumps" the count of the number of odd integers successfully subtracted from the square. The add is done by the LEAX 1,X instruction, which adds 1 to the contents of X and puts the result in X. We'll discuss in detail how this instruction works in the next chapter.

Two is subtracted from the odd integer in Y to get the next odd integer. We started off with +1, and after the first subtract we have -1. The instruction here is an LEAY -2,Y which we'll discuss in the next chapter. The result in Y is stored in RAM location $3002.

An ADDD $3002 is done. This "subtracts" an odd integer in $3002 from the square (or from the "residue" of the last subtract) in HL.

The Carry Condition Code is set to 1 (C) if the result of this subtract is 0 or greater. In other words, as long as the result in D is not a negative number, the C Condition Code will be set. As soon as the result "goes negative," the C Condition Code will be reset (NC or 0). Why is the Carry set in this fashion? There's no easy answer to this. The Carry is simply set from the carry bit out of D as the add is done, and one of the peculiarities of an add is that there will be a carry as long as the result is not negative. You don't have to memorize this fact, but bear in mind that the Carry can be used to test for this condition.

The instruction used to test the Carry is "Branch on Carry Set" or BCS, which is the same as "Branch if C=1." If the result has not "gone negative," the BR is made back to REL010 for the next operation. If the result has gone negative, X contains the count of odd integers successfully subtracted, and this is put into locations $3003 and $3004.

Hints and Kinks 8-2
Why the Carry Flag is Set

The best way to see how the carry flag works is to try some ADD examples yourself. You'll see that there is **always** a carry until the result "goes negative." Examples:

```
  00000100     00000011     00000010     00000001     0000000
 +11111111    +11111111    +11111111    +11111111    +1111111
 ----------   ----------   ----------   ----------   ----------
C 00000011   C 00000010   C 00000001   C 00000000     1111111
```

If you like, enter and assemble the program, and then single step through the program. You'll be able to see what is happening in the loop quite easily. Also, you might want to refer to Figure 8-2, a "flowchart" of the program. It represents the program "flow" in schematic form.

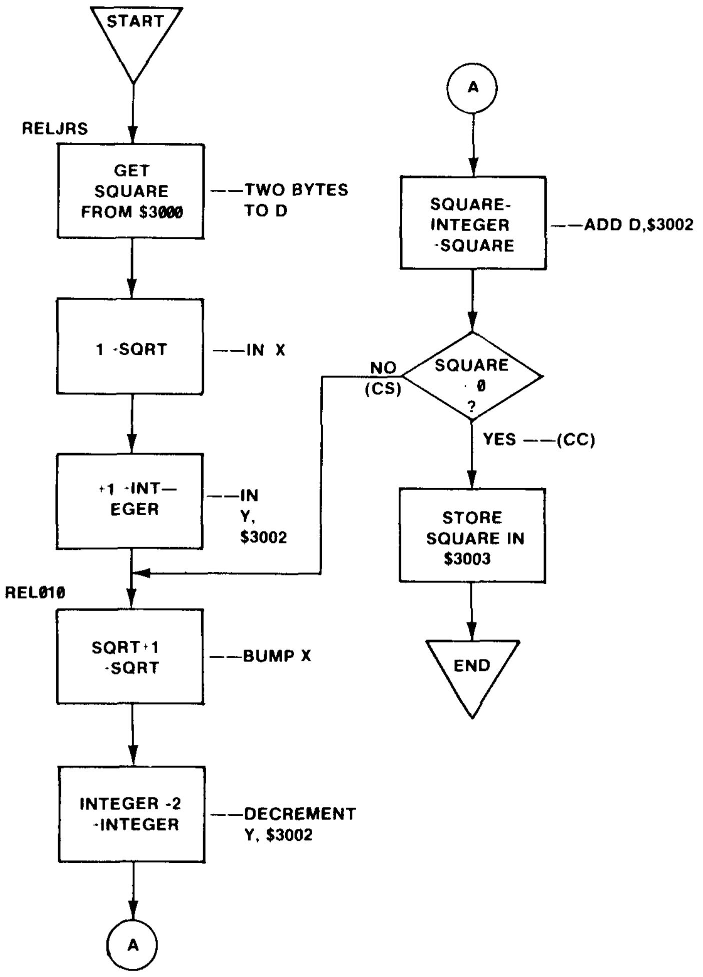


Figure 8-2. Square Root Flowchart

Now for the reason for this chapter, the format of "Relative" Branch instructions. After assembly, look at the machine-code for the BCS. It should be 25 F3 in hexadecimal. Notice anything different about the BR codes compared to the JMP of the last chapter?

The BCS instruction is 2 bytes long, while the JMP is 3 bytes! And that's the reason BR instructions were added to the 6809 instruction set, primarily to save memory. As jumps are used all the time, saving one byte in a jump can result in a 5% or more savings in memory space for a program.

Relative Addressing

Look again at the BCS bytes. The first byte is an "opcode" byte that tells the 6809 that a BR is about to be executed. The next byte somehow must specify the address for the jump. But how?

The BR uses "relative addressing." Relative to what? Relative to the location of the instruction. Here's the way the 6809 finds the address for the jump (see Figure 8-3):

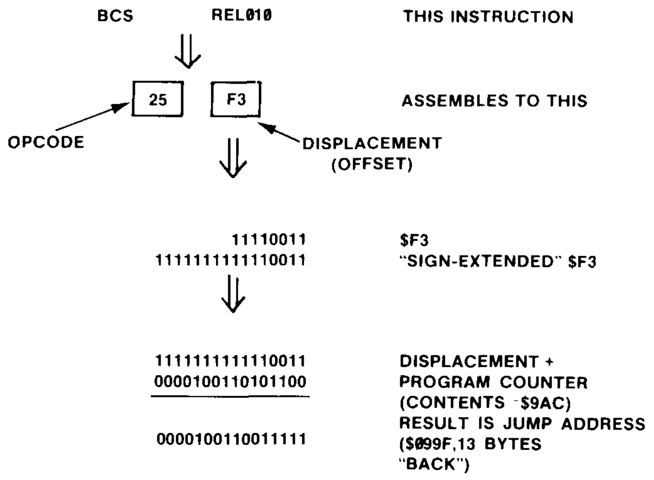


Figure 8-3. Relative Addressing Example

The second byte of the BCS instruction is made into a 16-bit number. This byte is an 8-bit "signed value" (negative or positive), with the first bit representing the sign. Our old friend (enemy?) the two's complement representation is in force here.

To make an 8-bit two's complement number into 16 bits, we have to "sign extend" the sign bit. If the sign bit is a 0, we put zeroes into the most significant byte; if the sign bit is a 1, we put ones into the most significant byte. In this case we'd need ones.

This 16-bit number is then added to the contents of the Program Counter register. The Program Counter always points to the next instruction to be executed. In the case of the BCS, it points to the STX $3003 instruction.

Adding the PC and the 16-bit number together gives the location of the branch address, as shown in the figure. And all that with just 8 bits for an address!

Of course, all of this is done internally in the 6809. You never have to do the arithmetic as we just did. The assembler will take care of putting in the proper value in the Branch instruction, so that all you have to do is use a symbolic label for the branch location, as we did.

Types of BRs

There are 17 conditional BRs, and one unconditional BR. Some of the conditional BRs were covered in the last chapter. The 17 conditional BRs are

Mnemonic	Meaning	CC For Branch
BEQ	Branch if Equal (Zero)	Z=1
BNE	Branch if Not Equal (non-Zero)	Z=0
BCS	Branch if Carry Set	C=1
BLO	Branch if Lower	C=1
BCC	Branch if Carry Clear	C=0
BHS	Branch if High or Same	C=0
BMI	Branch if Minus	N=1
BPL	Branch if Plus	N=0
BVS	Branch if Overflow	V=1
BVC	Branch if No Overflow	V=0
BLT	Branch if Less Than	N XOR V=1
BLE	Branch if Less Than or Equal	Z=1 or N XOR V=1
BGE	Branch if Greater Than or Equal	Z=1 or N XOR V=0
BGT	Branch if Greater Than	N XOR V=0
BLS	Branch on Low or Same	Z=1 or C=1
BHI	Branch if Higher	Z=0 and C=0
BRN	Branch Never	NOP
BRA	Branch Always	Unconditional

Hinks and Kinks 8-3
What is the BRN?

The BRN, or Branch Never, is not a joke. It is the same as a "NOP" instruction that does nothing except take up space. The BRN takes up 2 bytes and the LBRN takes up 3 bytes. The NOP uses a different "opcode" and takes up 1 byte. All three instructions leave the

Condition Codes unchanged and do not modify any cpu registers or memory locations. Why use these 3 NOPs? As "time wasters" in timing loops or as "padding" for possible "patches" to object code at a later time.

The above list needs some explanation. The conditional branches are usually used after a CMP or SUB. When used this way it's important to determine whether you're comparing two absolute numbers or two two's complement numbers.

Here are the Branches to be used for comparing two absolute (unsigned) numbers:

Branch For	Use
A<B	BLO
A<=B	BLS
A=B	BEQ
A>=B	BHS
A>B	BHI
A< >B	BNE

Here are the Branches to be used for comparing two signed (two's complement) numbers:

Branch For	Use
A<B	BLT
A<=B	BLE
A=B	BEQ
A>=B	BGE
A>B	BGT
A< >B	BNE

Note the BRA instruction in the main list. This is a Relative unconditional branch that should usually be used in lieu of the JMP instruction, subject to the restrictions we're going to discuss in the next topic. Also note the "BRN," Branch Never, an instruction that is the same as a "do nothing" or "NOP" (no operation) instruction.

Limitations of BRs

There is one slight hitch in using BRs. As there is only one byte for the address, the jump "range" is limited. You know that in an 8-bit two's complement number we can hold values of -128 through +127. As the second byte is the "relative" jump address in two's complement form, we can therefore only jump 128 locations back and 127 locations forward.

And don't forget that those numbers are referenced to the Program Counter, which points to the instruction after the BR. Referenced to the BR, then, we can only jump back 126 bytes or forward 129 bytes.

If we try to use a BR and jump "out of the range," EDTASM+ will detect the error and give us an error message of 'BYTE OVERFLOW.' What can we do about this situation?

The Long Branch is *Not* a Fort Worth Ranch

Each of the Branches discussed above has an alternate form, the "Long Branch." This form is shown in Figure 8-4. The Branch is still "Relative" to the contents of the Program Counter, but now there are 4 bytes instead of two in the instruction. If we were to use a Long Branch BCS in the square root program, we'd see the listing shown in Figure 8-5.

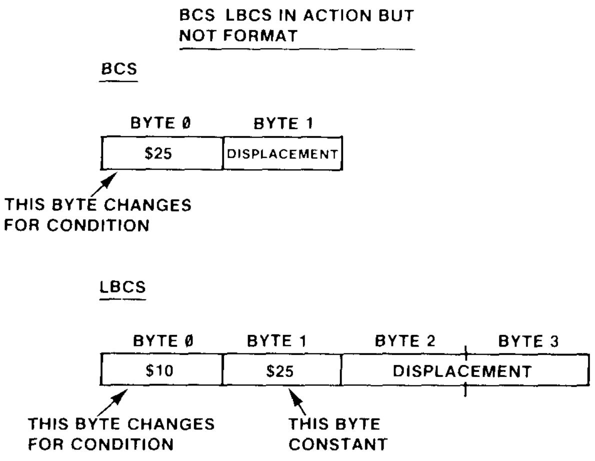

BCS LBCS IN ACTION BUT NOT FORMAT
BCS
BYTE Ø
BYTE 1
$25
DISPLACEMENT
THIS BYTE CHANGES FOR CONDITION
LBCS
BYTE Ø
BYTE 1
BYTE 2
BYTE 3
$10
$25
DISPLACEMENT
THIS BYTE CHANGES FOR CONDITION
THIS BYTE CONSTANT

Figure 8-4. Long Branch Format

```
                     00100 * RELATIVE BRANCHES
0992 FC   3000       00110 RELJRS  LDD    $3000   LOAD SQUARE
0995 8E   FFFF       00120         LDX    #-1     CLEAR SQUARE ROOT
0998 108E 0001       00130         LDY    #1      INITIALIZE ODD INTEGER
099C 10BF 3002       00140         STY    $3002   STORE IN MEMORY VARIABLE
09A0 30   01         00150 REL010  LEAX   1,X     SQUARE ROOT+1
09A2 31   3E         00160         LEAY   -2,Y    ODD INTEGER -2
09A4 10BF 3002       00170         STY    $3002   STORE ODD INTEGER
09A8 F3   3002       00180         ADDD   $3002   SQUARE-ODD INTEGER
09AB 1025 FFF1       00190         LBCS   REL010  LOOP IF NOT MINUS
09AF BF   3003       00200         STX    $3003   STORE SQUARE ROOT
09B2 7E   09B2       00210 LOOP    JMP    LOOP    LOOP HERE
          0000       00220         END
00000 TOTAL ERRORS

LOOP     09B2
REL010   09A0
RELJRS   0992
```

Figure 8-5. Long Branch Program

The Branch would be made exactly the same, except that the instruction would take more time to execute and would take up more space in memory.

Use the Long Branch in place of the BR instruction when the Branch will be "out of range." Don't use a Long Branch if you're uncertain about the range, as EDTASM+ will let you know during assembly, and you can do a quick edit and reassembly with a Long Branch in place of the Branch.

Either a Branch or Long Branch, therefore, can be used to Branch anywhere in memory, based on the condition of the Compare, Subtract, or **other instruction** preceding the Branch.

Review

To review what we've learned in this chapter:

- Relative branches are 2 bytes instead of 3 bytes as in the JMP, saving memory space
- Relative branches use a two's complement one-byte relative address which will give the jump address when added to the Program Counter
- There are 17 conditional BRs and one unconditional BRA; the conditional BRs work with Condition Code settings of various combinations
- The range of a BR is limited to 126 locations back or 129 locations forward from the BR; this is usually enough
- A Long Branch BR uses a 16-bit displacement to enable relative branches anywhere in memory; Long Branches can be used for any BR instruction

For Further Study

Branch and Long Branch instructions (Appendix II)

KEY CHART — CHAPTER 9

INSTRUCTIONS

ABX	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
ADCA	**BITA**	CLR	~~LDU~~	RORA
ADCB	**BITB**	*CMPA*	~~LDX~~	RORB
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
~~ADDB~~	~~LBLE~~	~~CMPD~~	**LEAS**	RTI
~~ADDD~~	~~BLO~~	~~CMPS~~	**LEAU**	RTS
ANDA	~~LBLO~~	~~CMPU~~	**LEAX**	SBCA
ANDB	~~BLS~~	~~CMPX~~	**LEAY**	SBCB
ANDCC	~~LBLS~~	~~CMPY~~	LSLA	SEX
ASLA	~~BLT~~	**COMA**	LSLB	~~STA~~
ASLB	~~LBLT~~	**COMB**	LSL	~~STB~~
ASL	~~BMI~~	**COM**	LSRA	~~STD~~
ASRA	~~LBMI~~	CWAI	LSRB	~~STS~~
ASRB	~~BNE~~	DAA	LSR	~~STU~~
ASR	~~LBNE~~	**DECA**	MUL	~~STX~~
~~BCC~~	~~BPL~~	**DECB**	**NEGA**	~~STY~~
~~LBCC~~	~~LBPL~~	**DEC**	**NEGB**	~~SUBA~~
~~BCS~~	~~BRA~~	**EORA**	**NEG**	~~SUBB~~
~~LBCS~~	~~LBRA~~	**EORB**	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	**ORA**	SWI
~~LBEQ~~	~~LBRN~~	**INCA**	**ORB**	SWI2
~~BGE~~	BSR	**INCB**	**ORCC**	SWI3
~~LBGE~~	LBSR	**INC**	PSHS	SYNC
~~BGT~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~LBGT~~	~~LBVC~~	JSR	PULS	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
DISPLACEMENT INDEXED
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
FCB	RMB
FCC	SET
FDB	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
INCREMENTS/DECREMENTS	VARPTR USE
COMPLEMENTS	ROM SUBROUTINES
LOGICAL OPERATIONS	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 9
Increments, Decrements, Complements, and Logical Operations

Increments and decrements are used quite often in assembly-language programs — they add or subtract 1 count. Complements include the COM and NEG instructions, which take either the "ones" or "two's" complement. Logical operations are similar to BASIC ANDs and ORs.

Increments and Decrements

DEC stands for "Decrement," and INC stands for "Increment." An increment adds one to the contents of the A or B registers or memory location, while a decrement subtracts one from the contents of the A or B registers or memory location.

Increments and decrements exist because adding 1 or subtracting 1 is a very common operation in assembly-language code. In the case of using the A register, for example, it's much better to do an INCA then to do an ADDA #1. The INCA is a one-byte instruction while the ADDA #1 is 2 bytes in length.

The mnemonics for INC and DEC are shown in this code:

```
INCA            INCREMENT A
INCB            INCREMENT B
INC    $3000    INCREMENT
                LOCATION $3000
DECA            DECREMENT A
DECB            DECREMENT B
DEC    $3000    DECREMENT
                LOCATION $3000
```

Eight-bit increments and decrements affect the Condition Codes about the same way that ADDs and SUBs affect the CCs. A zero result after a decrement sets the Z Condition Code, for example. We said "about the same way" because neither the Carry (C) Condition Code nor the Overflow Condition Code (V) are affected after an increment or decrement; they remain the same as they were before the INC or DEC.

The LEA for 16-Bit Registers

What about 16-bit registers such as X and Y? Can they be incremented or decremented? There's a special instruction for doing this and other manipulations on 16-bit registers called "Load Effective Address," or LEA. The LEA mnemonic is suffixed by either an S, U, X, or Y for four operations on the four registers:

```
LEAS
LEAU
LEAX
LEAY
```

LEA instructions work like this: The contents of either the S, U, X, or Y register is added together with an 8-bit or 16-bit signed displacement for the LEA instruction in a manner very similar to relative branches. The result is loaded into the specified register. This is harder to describe than use:

```
LEAX     -1,X        Add (X)+(-1), result to X
LEAY     -1,X        Add (X)+(-1), result to Y
LEAS     -3,S        Add (S)+(-3), result to S
LEAX     +10000,X    Add (X)+(+10000), result to X
```

Do you see how this works? In the simplest case, it's a replacement for INC and DEC for a 16-bit register, as in LEAX -1,X, which decrements X. In a more complicated form, it can add up to $FFFF or subtract up to -$8000 from a 16-bit register and put the result into the same or different 16-bit register.

The only Condition Code affected is Z, when an LEAX or LEAY is done. For all other LEAs the Condition Codes remain unchanged.

We'll see more use of this in later chapters.

Hints and Kinks 9-1
More on the LEA Instruction

The predecessor of the 6809, the 6800 microprocessor, had an "increment index" and "decrement index" instruction to modify the contents of X by +1 or -1. The 6809 goes another order of magnitude better than this and allows you to increment or decrement X, Y, S, or U by any 16-bit value.

The number of bytes in the "displacement" is dependent upon the size of the displacement. If the displacement is less than +128 or greater than -128, EDTASM+ generates a one-byte displacement. If the displacement is greater than +127 or less than -128, EDTASM+ generates a 2-byte displacement. You, as a programmer, don't have to worry about the instruction size — the assembler will take care of it automatically.

The NEG and COM Instructions

The NEGate and COMplement instructions are two instructions that operate on data in the A or B registers or an 8-bit memory location.

The COM instruction takes the contents of the A register and "one's complements" it, by changing all ones to zeroes and all zeroes to ones.

Suppose that the A register contained

00111111

After a COMA instruction had been done, the new contents would be:

11000000

The COM instruction changes the N (negative) and Z (Zero) Condition Codes on the result of the Complement. If 11111111 was complemented to 00000000, for example, the Z Condition Code would be set.

The NEG instruction takes the contents of the A or B register or a memory location and "two's complements" it, changing all ones to zeroes and all zeroes to ones and adding one. We've already seen how the two's complement works in an earlier lesson. The NEG simply performs the two's complement conversion automatically. The effect of this is to "negate" a number, changing a positive number into a negative number and vice versa.

Suppose that the B register contained:

00111111, or decimal 63

After a NEGB, the A register would contain:

11000001,

or decimal -63 (in two's complement).

The NEG sets the Condition Codes just as in a SUB instruction. If the result was 10000000, for example, the Z Condition Code would be reset (NE) and the N Condition Code would be set (MI).

The COM and NEG are used infrequently compared to adds, subtracts, decrements, and increments.

Logical Operations

The 6809 has three instructions that perform logical operations, the OR, AND, and exclusive OR (EOR).

ORs

If you've done some BASIC programming, you'll be familiar with the first two logical operations, and the EOR is simply a variation.

The AND and OR can operate on the A or B registers and **also the Condition Codes.** The EOR works only on A or B.

An OR takes the 8-bit value in A, B, or the CC and ORs it with the second operand from memory or an immediate operand. An OR operates on a "bit-level" — one bit at a time with no bit affecting any other bit. For each bit:

```
0 OR 0=0
0 OR 1=1
1 OR 0=1
1 OR 1=1
```

A bit in the result is set, then, when either one OR the other bit OR both is set. The result of a typical OR might be:

```
   00111010
OR 01010111
   --------
   01111111
```

Only in the case of the most significant bit was the result bit not 0, and that was because both operand bits were 0.

ORs are typically used to set a single bit in the middle of other bits. Suppose you wanted to set bit 5 of the A register. (Bit positions are 7, 6, 5, 4, 3, 2, 1, and 0 from left to right.) One way to do it would be:

```
ORA   $20          SET BIT 5
```

ANDs

An AND takes the 8-bit value in A, B, or CC and ANDs it with the second operand. An AND also operates on a "bit-level" — one bit at a time with no bit affecting any other bit. For each bit:

```
0 AND 0=0
0 AND 1=0
1 AND 0=0
1 AND 1=1
```

A bit in the result is set, then, when one bit AND the other bit are set. The result of a typical AND might be:

```
    00111010
AND 01010111
    --------
    00010010
```

The only result bits that were set were ones in which both operand bits were ones.

ANDs are typically used to "mask" data. Suppose you wanted to find the setting of bits 1 and 0 of the B register. One method would be:

```
ANDB  #3           GET BITS 1 AND 0
```

At the end of these two instructions, B would contain 000000XX, where XX are the settings of bits 1 and 0 in the B register. The bits in bit positions 1 and 0 "fell" through on the AND, and the other bits were "masked out":

```
    01010010    (B register)
AND 00000011    (AND value)
    --------
    00000010    (result of AND)
```

EORs

An EOR takes the 8-bit value in A or B and EORs it with the second operand from memory or an immediate operand. An EOR again operates on a "bit-level" — one bit at a time with no bit affecting any other bit. For each bit:

```
0 EOR 0=0
0 EOR 1=1
1 EOR 0=1
1 EOR 1=0
```

A bit in the result is set, then, when either one OR the other bit but NOT both bits is set. The result of a typical EOR might be:

```
    00111010
EOR 01010111
    --------
    01101101
```

Whenever the operand bits were both 0s or 1s, the result bit was a zero.

EORS are used infrequently compared to ORs and ANDs. One classic example of the use of an EOR is to check the sign of a result. Suppose that we were going to multiply two 8-bit numbers. The sign of the result could be determined by:

```
    10101010    (-86 decimal)
EOR 01010100    (+84 decimal)
    --------
    11111110    (EOR result)
```

The result sign of the EOR is a 1, so the sign of the multiply result will be a 1, or negative.

Using ORCC and ANDCC

OR and AND can also be used to set or reset 1 or more Condition Codes. (Refer to Figure 6-1.)

To set any Condition Code use the OR:

```
ORCC  #1        SET CARRY
ORCC  #5        SET ZERO, CARRY
```

To reset any Condition Codes, use a "mask" type immediate value. To reset the Carry, for example, use the mask complement of 1, $FE (11111110):

```
                    ANDCC #$FE        RESET CARRY
```

Use a similar type of mask to reset more than one Condition Code:

```
                    ANDCC #$FA        RESET Z,C (11111010)
```

A Special AND

There're another two instructions that we should mention here, the BITA and BITB instructions. These instructions are the same as ANDs for A and B except that the result is not put back into A and B, but dropped into the bit bucket. The Condition Codes are set the same as in an ANDA or ANDB. BITA and BITB can be used with an immediate operand or an operand from memory.

```
            BITA    #5        TEST BITS 2 AND 0 (Z,CC)
            BIT     $3000     AND B WITH ($3000) FOR TEST
```

Review

To review what we've learned in this chapter:

- Increments and decrements add or subtract 1 from A or B or from a memory location
- Condition Codes are set for 8-bit increments and decrements, except for H and C
- The LEA instruction adds or subtracts a signed displacement to the X, Y, U, or S register
- The COM "complements" data, changing all 0s to 1s and all 1s to 0s
- The NEG takes the two's complement, or negation, of data
- ANDs, ORs, and EORs work with one operand in A, B, or CC (AND or OR) and one from an immediate operand or memory
- ANDs set each result bit to a 1 only if both operand bits are 1
- ORs set each result bit to a 1 if either operand bit is a 1
- EORs set each result bit to a 1 if either but not both operand bits is a 1

KEY CHART — CHAPTER 10

INSTRUCTIONS

	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
A	~~BITA~~	CLR	~~LDU~~	RORA
B	~~BITB~~	~~CMPA~~	~~LDX~~	RORB
~~A~~	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
~~B~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	RTI
~~D~~	~~BLO~~	~~CMPS~~	~~LEAU~~	RTS
~~A~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	**SBCA**
~~B~~	~~BLS~~	~~CMPX~~	~~LEAY~~	**SBCB**
~~CC~~	~~LBLS~~	~~CMPY~~	LSLA	SEX
A	~~BLT~~	~~COMA~~	LSLB	~~STA~~
B	~~LBLT~~	~~COMB~~	LSL	~~STB~~
	~~BMI~~	~~COM~~	LSRA	~~STD~~
A	~~LBMI~~	CWAI	LSRB	~~STS~~
B	~~BNE~~	DAA	LSR	~~STU~~
[illegible]	~~LBNE~~	~~DECA~~	MUL	~~STX~~
~~C~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~CC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~S~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~CS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~Q~~	~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~EQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~E~~	BSR	~~INCB~~	~~ORCC~~	SWI3
~~GE~~	LBSR	~~INC~~	PSHS	SYNC
~~T~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~GT~~	~~LBVC~~	JSR	PULS	~~TSTA~~
—	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~H~~	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
~~S~~	~~CLRA~~	~~LDD~~	ROLB	

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	U MOVE BLOCK
~~D(ISPLAY)~~	V (ERIFY) BLOCK
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	· BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ FORCE NUMERIC,BYTE
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

ADDRESSING MODES

~~HERENT~~
~~RECT~~
~~TENDED~~
~~MEDIATE~~
~~MPLE INDEXED~~
~~LATIVE~~
SPLACEMENT INDEXED
TO INCREMENT/DECREMENT
DIRECT
PHISTICATED

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
MULTIPLE PRECISION	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

PSEUDO OPS

U	ORG
B	RMB
C	SET
B	~~SETDP~~

d Type **Present Chapter**
ular Type Future Chapters
~~ic Type~~ *Past Chapters*

Chapter 10
Using the Carry for Gobs of Precision

You might be wondering about the value of the Carry Condition Code. If you were, we'll show you in this chapter how that innocuous Carry can be used in "multiple-precision" operations to string together 8-bit adds and subtracts that can process infinitely large numbers. We'll look at ADC and SBC, adds and subtracts with "Carry." These instructions are very similar to the standard ADDs and SUBs, except that the state of the Carry Condition Code is added or subtracted in. ADC and SBC allow "multiple-precision" operations which can extend the range of processing to any size number, not just 8 and 16 bits.

Multiple-Precision Numbers

A multiple-precision number is a fancy term for any integer number format that is larger than the size the microprocessor can handle with its built-in instructions.

In the 6809, we can add 8 and 16-bit operands. The maximum number that can be represented in 8 bits is 255 (unsigned), while the maximum number that can be represented in 16 bits is 65,535 (unsigned). What about large numbers?

One way to handle large numbers is to use "floating-point" numbers. Floating-point representation is what BASIC uses to handle single-precision and double-precision numbers. Floating-point operations are rather complex, however, and beyond the scope of this book.

Hints and Kinks 10-1
Microcomputer Math

See my Howard W. Sams book "Microcomputer Math" for more on floating-point number format in microcomputers.

There's no reason we can't handle any size number in the 6809. We may have to string the numbers together as a series of bytes, but we can easily handle 4-byte or 8-byte numbers.

Hints and Kinks 10-2
How Many Bytes for How Much Precision?

As a rule of thumb, one decimal digit can be held in 3 1/2 bits. In 8 bits, for example, you can hold decimal values from 0 through 255, or about 2.55 decimal digits (8/3.5=2.28). In 16 bits you can hold decimal values from 0 through 65,535, or about 4.6 decimal digits (16/3.5=4.57).

In 4 bytes, then, you'd have 32 bits, or about 32/3.5=9.2 decimal digits. In 8 bytes you'd have 32/3.5 or about 18 decimal digits. You can see that it doesn't take too many bytes to get a great deal of precision.

Look at Figure 10-1. In this figure, we've got a 4-byte number. In four bytes, we can represent 2 to the 32nd power or about 4,295,000,000. That's not an unreasonable number range to work with, even for Federal budget deficits. As a matter of fact, we can get more **precision** than we can get in single-precision BASIC. Note that I said more precision, which essentially means more digits; we still don't have the range of BASIC variables which also allow **exponents** such as 1.234 X 10 to the 14th power (1.234E+14).

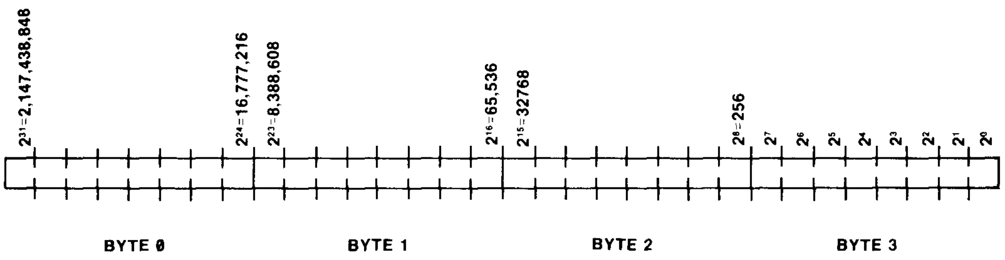


Figure 10-1. Four-Byte Multiple Precision Number

If we want even more of a range, we can go to a larger number of bytes, but we'll consider 4 bytes here, for convenience.

The format of multiple-precision numbers is about the same as 8- or 16-bit numbers. The first bit may or may not be a sign bit, depending upon whether you're working with absolute or two's complement numbers. The only real difference is that the number is spread out among several bytes, and that adds, subtracts, and other operations have to be handled in 8-bit or 16-bit "chunks."

Suppose that we want to add the numbers shown in Figure 10-2. The two 4-byte numbers here represent 8,000,001 and 8,777,215. The first add adds the bytes $01 and $FF, hexadecimal. The next add adds the next two bytes and any carry from the least significant byte. The next add adds the third bytes of the operands and any carry from the second add. Finally, the last add adds the fourth bytes of the operands and any carry from the third add.

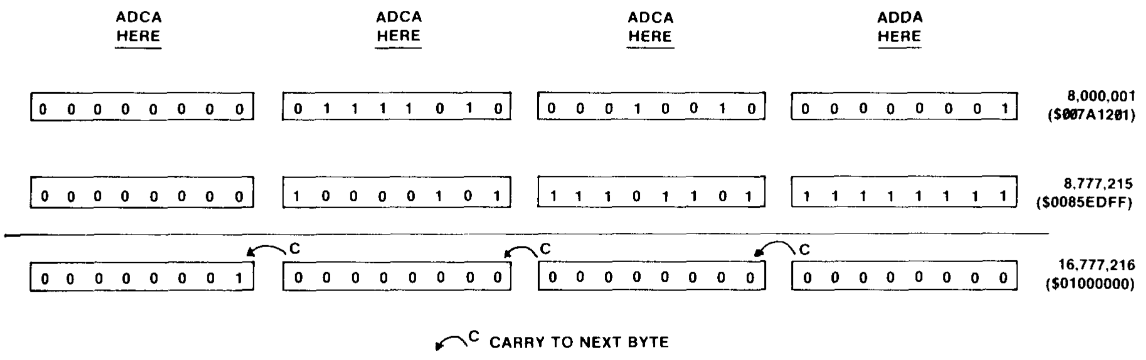


Figure 10-2. Adding Multiple Precision Numbers

The first add is our old friend ADDA, an 8-bit add. The remaining three adds are ADCA, or Adds with Carry so that any carry from the low-order is added in.

Eight-Bit Add With Carry

The 8-bit ADCs operate very similarly to the standard 8-bit add. An immediate value or contents of a memory location is added to the contents of the A or B registers, with the result going back into the A or B register. However, in addition to the second operand being added to the accumulator, the current state of the Carry Condition Code is also added in. As the Carry Condition Code may be set or reset, the add results in either the same sum as a normal ADD, or a result that is one greater than the normal ADD.

Let's look at a program that performs the 4-byte multiple-precision add that we diagrammed above:

```
                * ADC FOR MULTIPLE-PRECISION 4-BYTE ADDS
09B9 8E   3003  MPADDS  LDX    #$3003   POINT TO OP1+3 BYTES
09BC 108E 3007          LDY    #$3007   POINT TO OP2+3 BYTES
09C0 C6   04            LDB    #4       LOOP COUNTER
09C2 1C   FE            ANDCC  #$FE     CLEAR CARRY
09C4 A6   84    MPA010  LDA    ,X          GET OPERAND 1
09C6 A9   A4            ADCA   ,Y          ADD IN OPERAND 2
09C8 A7   84            STA    ,X          STORE RESULT
09CA 30   1F            LEAX   -1,X        DECREMENT OP1 PNTR
09CC 31   3F            LEAY   -1,Y        DECREMENT OP2 PNTR
09CE 5A                 DECB               DECREMENT COUNT
09CF 26   F3            BNE    MPA010      LOOP IF NOT 4
09D1 7E   09D1  LOOP    JMP    LOOP     LOOP HERE
          0000          END
00000 TOTAL ERRORS

LOOP    09D1
MPA010  09C4
MPADDS  09B9
```

Figure 10-3. Multiple-Precision Add Program

If you'd like to assemble the code, you'll be able to see what's happening very easily.

The four bytes at $3000 through $3003 hold the first operand. The four bytes at $3004 through $3007 hold the second operand. After the add is done, the result will replace the first operand at locations $3000 through $3003. A suggested first set of operands is:

Operand 1	($3000-$3003)	$00 $80 $27 $FF	+8,398,847 decimal
Operand 2	($3004-$3007)	$83 $A0 $7A $11	-2,086,634,991 decimal
Result	($3000-$3003)	$84 $20 $A2 $10	-2,078,236,144 decimal

If you've assembled the program, use the slash command of ZBUG to set up the operands. Breakpoint at LOOP and then execute from MPADDS.

When the breakpoint at LOOP is reached, you can use the ZBUG T command to look at the results in the operand 1 area ($3000 through $3003). Use T3000 3003.

The program works like this: X is used as a pointer to the first operand area. Since we'll be adding bytes **from least significant byte to most significant,** X is set to point to the last byte of the first operand.

Y is used as a pointer to the second operand area. It is initially set to point to the last byte of the second operand.

The B register is used as a loop counter for the 4 adds that will take place.

The loop starts at MPA010. Each time through the loop, X and Y point to the next set of bytes in both operands.

The A register is loaded with the first operand byte by the LDA ,X, which uses the X register to load the byte from the operand 1 area. The ADC instruction then adds the second operand byte, using the Y register as a pointer. The result in A is then stored back in the first operand area. The ADC adds the two bytes, but also adds in any Carry from the previous add.

Important note: The only instruction affecting the Carry in the loop is the ADC, therefore the Carry always holds the Carry from the last ADC.

Hints and Kinks 10-3
When Is the Carry Affected?

You can easily see for what instructions the Carry is affected by going down the list of instructions in Appendix II. The Carry is changed for adds, subtracts, compares, complements, negates, and shifts, including rotates. The Carry Condition Code, then, is very much geared to "arithmetic operations."

The Carry before the first add was reset to 0 by the ANDCC instruction, which used the immediate value of $FE as a mask to reset the Carry. The first ADC, then, uses a carry of 0.

Experiment with different 4-byte operands, and you'll get a good idea of how this multiple-precision add works.

Eight-Bit Subtracts With "Borrow"

The 8-bit subtract, SBC, is identical to the 8-bit ADC, except for the actual operation, of course. The SBC again will work only with the A or B registers, and not with any 16-bit register.

In the SBC, any Carry from a lower order is actually a "Borrow," but the borrow is termed a Carry for convenience.

To see that the subtract does indeed work look at the program below. It duplicates the earlier program, except that the ADC is replaced by an SBC. The operands are 4-byte operands in the $3000 through $3007 area as before.

```
                        00100 * SBC FOR MULTIPLE-PRECISION 4-BYTE SUBTRACTS
09C0 8E   3003          00110 MPSUBS  LDX    #$3003    POINT TO OP1+3 BYTES
09C3 108E 3007          00120         LDY    #$3007    POINT TO OP2+3 BYTES
09C7 C6   04            00130         LDB    #4        LOOP COUNTER
09C9 1C   FE            00140         ANDCC  #$FE      CLEAR CARRY
09CB A6   84            00150 MPS010  LDA    ,X          GET OPERAND 1
09CD A2   A4            00160         SBCA   ,Y          SUBTRACT OPERAND 2
09CF A7   84            00170         STA    ,X          STORE RESULT
09D1 30   1F            00180         LEAX   -1,X        DECREMENT OP1 PNTR
09D3 31   3F            00190         LEAY   -1,Y        DECREMENT OP2 PNTR
09D5 5A                 00200         DECB               DECREMENT COUNT
09D6 26   F3            00210         BNE    MPS010      LOOP IF NOT 4
09D8 7E   09D8          00220 LOOP    JMP    LOOP      LOOP HERE
          0000          00230         END
00000 TOTAL ERRORS

LOOP     09D8
MPS010   09CB
MPSUBS   09C0
```

Figure 10-4. Multiple-Precision Subtract Program

If you assemble the program and execute it as before try these operands:

Operand 1 ($3000-$3003) $00 $80 $27 $FF +8,398,847 decimal

Operand 2 ($3004-$3007) $83 $A0 $7A $11 (-) -2,086,634,991 decimal

Result ($3000-$3003) $7C $DF $AD $EE +2,095,033,838 decimal

The result will be in the $3000 area as before. These operands cause a borrow from the next higher byte, and you can see how the SBC uses this borrow in the subtract of the current byte.

We'll leave it up to you to experiment with this multiple-precision subtract with other operands.

Other Multiple-Precision Operations

You've seen in the above discussion how almost any size number can be handled for adds and subtracts by using ADC and SBC. But what about multiplies, divides, and other operations? Generally these are quite a bit more difficult. We'll be looking at some multiply and divide programs in a later chapter and we'll discuss some possibilities for multiple-precision operations in that chapter.

Review

To review what we've covered in this chapter:

- Multiple-precision numbers are numbers that use multiple bytes to provide a larger number range than is possible in 8 or 16 bits
- Multiple-precision numbers can be absolute or signed (two's complement) and resemble 8- or 16-bit numbers of these types
- Eight-bit adds with Carry operate similarly to the normal 8-bit adds, except that the current state of the Carry is added in
- Eight-bit subtracts with Carry operate identically to an 8-bit ADC, except that a "borrow" is subtracted from the A register along with the second operand; the borrow is held in the Carry Condition Code

For Further Study

Look at the Condition Code settings for the ADCs and SBCs (Appendix II)

KEY CHART — CHAPTER 11

INSTRUCTIONS

	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
~~A~~	~~BITA~~	CLR	~~LDU~~	RORA
~~B~~	~~BITB~~	~~CMPA~~	~~LDX~~	RORB
~~I~~	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
~~B~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	RTI
~~D~~	~~BLO~~	~~CMPS~~	~~LEAU~~	RTS
~~I~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~B~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~SG~~	~~LBLS~~	~~CMPY~~	LSLA	~~SEX~~
A	~~BLT~~	~~COMA~~	LSLB	~~STA~~
B	~~LBLT~~	~~COMB~~	LSL	~~STB~~
	~~BMI~~	~~COM~~	LSRA	~~STD~~
A	~~LBMI~~	CWAI	LSRB	~~STS~~
B	~~BNE~~	DAA	LSR	~~STU~~
	~~LBNE~~	~~DECA~~	MUL	~~STX~~
	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~G~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~S~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
	~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~Q~~	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~E~~	BSR	~~INCB~~	~~ORCC~~	SWI3
~~E~~	LBSR	~~INC~~	PSHS	SYNC
~~T~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~T~~	~~LBVC~~	JSR	PULS	~~TSTA~~
	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~H~~	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
~~B~~	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~ERENT~~
~~ECT~~
~~ENDED~~
~~EDIATE~~
~~PLE INDEXED~~
~~ATIVE~~
PLACEMENT INDEXED
TO INCREMENT/DECREMENT
IRECT
PHISTICATED

PSEUDO OPS

U	ORG
B	RMB
C	SET
B	~~SETDP~~

Type = Present Chapter
ular Type = Future Chapters
~~Type~~ = *Past Chapters*

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

A(SCII) DISPLAY	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	**U MOVE BLOCK**
~~D(ISPLAY)~~	**V (ERIFY) BLOCK**
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
L(OAD) ML FILE	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	+ **FORCE NUMERIC,BYTE**
P SAVE ML ON TAPE	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
DATA VALUES	SOUND
INDEXING	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

Chapter 11
Generating Data Values and Simple Indexing

In this chapter we'll discuss how to generate tables of data by using the FCB and FDB assembler "pseudo-ops." We'll also look at some of the simple "indexing" techniques which enable us to access tables. Another pseudo-op allows us to construct assembly-time strings of ASCII data similar to BASIC strings.

A List of Data

The simplest form of a table is simply a "one-dimensional" list of data, similar to a one-dimensional BASIC array. Let's build two versions of a data list, one with numeric data, and one with "alphanumeric" (text) data.

A Numeric Table Lookup

Suppose that we wanted to convert from degrees Centigrade to degrees Farenheit. One way to do it would be to use the formula:

F=(9/5)*C+32

where C is the temperature in degrees Centigrade and F is the temperature in degrees Farenheit. We could do a multiply, a divide, and an add, but another alternative would be to use a "look-up" table of Farenheit values. This table would appear as in Figure 11-1.

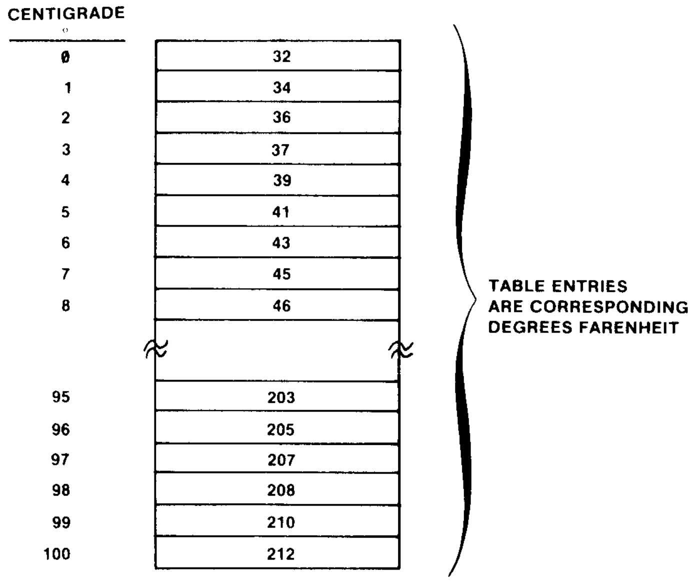


Figure 11-1. Lookup Table for Centigrade to Farenheit

The table is "accessed" by using the number of Centigrade degrees as an "index" value. The table starts with 0 degrees Centigrade and ends with 100 degrees Centigrade. To find the Farenheit degrees for 20 degrees Centigrade, for example, you'd look up the 21st "entry" in the table.

How would you construct such a table using Color Computer assembly language? To see the answer, look at this code:

```
* LOOKUP TABLE FOR CENTIGRADE TO
  FARENHEIT CONVERSION
* ONE ENTRY FOR EVERY DEGREE CENTIGRADE
CTOFTB  FCB     32      0 DEGREES C
        FCB     34      1
        FCB     36      2
        FCB     37      3
        FCB     39      4
        FCB     41      5
        FCB     43      6
        FCB     45      7
        FCB     46      8
        FCB     48      9
        FCB     50      10
        FCB     52      11
        FCB     54      12
        FCB     55      13
        FCB     57      14
        FCB     59      15
        FCB     61      16
        FCB     63      17
        FCB     64      18
        FCB     66      19
```

If you assemble this code, you'll see the listing shown in Figure 11-2, a list of 20 bytes, ranging from 32 ($20) through 66 ($42).

```
                    00100 * TABLE LOOKUP FOR C TO F CONVERSION
0A4B 8E   0A5A      00110 TABLOK  LDX     #CTOFTB LOAD TABLE START
0A4E F6   3000      00120         LDB     $3000   GET DEGREES C
0A51 3A             00130         ABX             ADD TO START
0A52 A6   84        00140         LDA     ,X      GET DEGREES F
0A54 B7   3001      00150         STA     $3001   STORE RESULT
0A57 7E   0A57      00160 LOOP    JMP     LOOP    LOOP HERE
                    00170 * LOOKUP TABLE FOR CENTIGRADE TO FARENHEIT
                    00180 * ONE ENTRY FOR EVERY DEGREE CENTIGRADE
0A5A      20        00190 CTOFTB  FCB     32      0 DEGREES
0A5B      22        00200         FCB     34      1
0A5C      24        00210         FCB     36      2
0A5D      25        00220         FCB     37      3
0A5E      27        00230         FCB     39      4
0A5F      29        00240         FCB     41      5
0A60      2B        00250         FCB     43      6
0A61      2D        00260         FCB     45      7
0A62      2E        00270         FCB     46      8
0A63      30        00280         FCB     48      8
0A64      32        00290         FCB     50      10
0A65      34        00300         FCB     52      11
0A66      36        00310         FCB     54      12
0A67      37        00320         FCB     55      13
0A68      39        00330         FCB     57      14
0A69      3B        00340         FCB     59      15
0A6A      3D        00350         FCB     61      16
0A6B      3F        00360         FCB     63      17
0A6C      40        00370         FCB     64      18
0A6D      42        00380         FCB     66      19
          0000      00390         END
00000 TOTAL ERRORS

CTOFTB   0A5A
LOOP     0A57
TABLOK   0A4B
```

Figure 11-2. Table Lookup Program 1

The FCB Pseudo-Op

The FCB is an assembler "pseudo-op" that does not generate an instruction, but generates "data" instead. Unlike BASIC, though, we have to be careful where we put the data. In BASIC the DATA statements are simply "in-line" with other BASIC statements. In assembly-language we can put data anywhere, but have to make certain that the program jumps around them.

You can see from the above code that you can use a label on a data area as well as an instruction. This allows you to symbolically reference the data, which, in this example, we've called CTOFTB, or "C to F Table." It also allows ZBUG to reference the data.

Each "entry" in this table is one byte long. Given a temperature reading in degrees Centigrade, we can easily find the equivalent degrees Farenheit by a "table lookup."

One program to do this is shown in the "code" of Figure 11-2.

This program takes a value in degrees Centigrade from memory location $3000, uses it as an index value, and "looks up" the corresponding degrees Farenheit in the CTOFTB.

X is used as a pointer, and is loaded with "CTOFTB." This symbol is the same as any other symbol used with an instruction. It is the symbolic name of the location, in this case a **data** location. The corresponding address of CTOFTB is put into the instruction as immediate data, as you can see in Figure 11-3.

```
                   00100 * TABLE LOOKUP FOR C TO F CONVERSION
0A4B 8E    0A5A    00110 TABLOK  LDX    #CTOFTB LOAD TABLE START
0A4E F6    3000    00120         LDB    $3000   GET DEGREES C
                     .
                     .
                     .
0A5A       20      00190 CTOFTB  FCB    32      0 DEGREES
                     .
                     .
```

Figure 11-3. Immediate Data Address

The B register is then loaded with the degrees Centigrade value from location $3000. This value is the "index value" for the table lookup. This value is then added to the table start value in X by the ABX instruction. The result in X points to the location in the table where the Farenheit byte will be found.

We haven't mentioned the ABX instruction before. It is a special add that adds the contents of the B accumulator to the contents of X and puts the result in X. No Condition Codes are affected. ABX is geared to the exact operation we're doing here.

The A register is then loaded with this value, and it's then stored in location $3001.

If you want to see how this works, assemble and execute the program as you've been doing in previous chapters, first putting a legitimate value in $3000 as the degrees Centigrade value and looking for the Farenheit result in $3001.

Entries of More Than One Byte

The table above was one of the simplest tables we could work with. Let's look at a more complicated table, one that uses "entries" greater than one byte, and also uses several "fields."

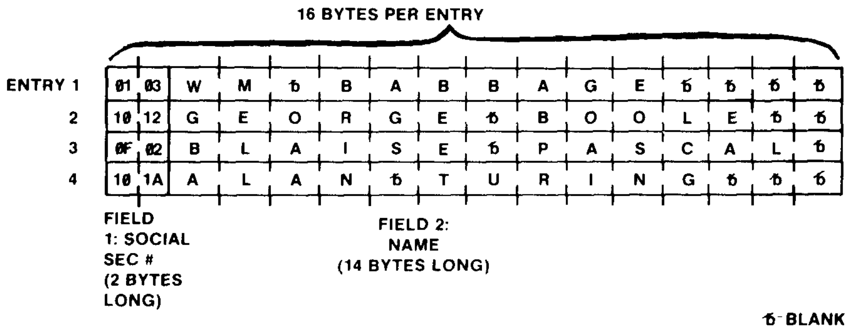

16 BYTES PER ENTRY
ENTRY 1 | 01 03 | W M ƀ B A B B A G E ƀ ƀ ƀ ƀ
2 | 10 12 | G E O R G E ƀ B O O L E ƀ ƀ
3 | 0F 02 | B L A I S E ƀ P A S C A L ƀ
4 | 10 1A | A L A N ƀ T U R I N G ƀ ƀ ƀ
FIELD 1: SOCIAL SEC # (2 BYTES LONG)
FIELD 2: NAME (14 BYTES LONG)
ƀ BLANK

Figure 11-4. Table with Large Entries - Structure

Each entry is made up of two "fields." A field is a subdivision of a "record," as we saw in an earlier lesson. The second field holds the name of the computer pioneer, and the first field holds his Social Security Number (they used shorter numbers in those days, of course). Each field is a "fixed length;" the second field is 14 bytes long, and the first field is 2 bytes long. The total length of each entry is 16 bytes.

This concept of entries in a table with fields within the entry could be expanded to tables with hundreds of entries and with many fields. For example, you might have a table of employees that had one field for an employee name, another for an employee number, another for marital status, and so on. Each entry in the table might be a hundred bytes long or so.

Why did we make the fields "fixed length" above? In fact, we could have used a "variable length" field, which would make the entries variable length also. Fixed-length entries are easier to work with, though, even though they do take up more space.

The FDB Pseudo-Op

This is the first time we've used the FDB pseudo-op. You'll recall that a pseudo-op is a command to the assembler and not an instruction mnemonic.

The FDB is used to build data, just as the FCB was used in the previous example. In this case the data for the FDB represents a 16-bit value. Instead of one byte being generated, as in the case of the FCB, two bytes will be produced. The mnemonic stands for "Form Double Byte."

Hints and Kinks 11-1
Multiple Operands for FCBs and FDBs

I hate to have to tell you this, but:

```
FCB    34,34,36,37
```

and:

```
FDB    1000,2000,131
```

are not legitimate assembly-language lines. You **must** have a separate FCB for every data byte and a separate FDB for every data "word" (2 bytes). One way to get around this somewhat is to pack 2 data bytes into an FDB:

```
FDB    $1234 = FCB  $12
               FCB  $34
```

The FCC Pseudo-Op

The FCC pseudo-op is also new. The FCC generates ASCII bytes. ASCII, of course, is a special code used to represent text data. All assemblers have such a pseudo-op to allow the programmer to easily construct messages and other text data. ASCII characters are shown in Appendix VI.

The FCC ("Form Constant Character") generates one ASCII byte for each character enclosed within the quotes. We've used a single quote around the characters here, but in fact, the "delimiter" could be any character that won't be used inside the string. EDTASM+ looks at the first character to define the "delimiter" and then looks for the second occurrence of the character to define the end of the string. Typical strings might be /STRING HERE WITH SLASH DELIMITER/, &STRING HERE WITH AMPERSAND DELIMITER&, =STRING HERE WITH EQUALS DELIMITER=.

Figure 11-5 shows the assembly of the table above. To see how this assembles in memory, use the A/IM option and then go to ZBUG.

```
                    00100 * TABLE WITH LARGE ENTRIES AND SEVERAL FIELDS
08DD      0103      00110 EMPTAB  FDB     $0103
08DF      57        00120         FCC     'WM BABBAGE    '
          4D
          20
          42
          41
          42
          42
          41
          47
          45
          20
          20
          20
          20
08ED      1012      00130         FDB     $1012
08EF      47        00140         FCC     'GEORGE BOOLE  '
          45
          4F
          52
          47
          45
          20
          42
          4F
          4F
          4C
          45
          20
          20
08FD      0F02      00150         FDB     $0F02
08FF      42        00160         FCC     'BLAISE PASCAL '
          4C
          41
          49
          53
          45
          20
          50
          41
          53
          43
          41
          4C
          20
090D      101A      00170         FDB     $101A
090F      41        00180         FCC     'ALAN TURING   '
```

Figure 11-5. Table With Large Entries - Program

```
                4C
                41
                4E
                20
                54
                55
                52
                49
                4E
                47
                20
                20
                20
                0000      00190             END
00000 TOTAL ERRORS

EMPTAB  08DD
```

Figure 11-5 continued

In ZBUG, set the ASCII examination option by using the A command:

```
*A/IM           (assemble)
*Z              (ZBUG)
#A              (set ASCII)
```

Now examine the locations from EMPTAB by using the slash or T commands. You'll see something like this:

```
#TABLE/         (DOWN ARROW)
 838/           (DOWN ARROW)
 839/     W     (DOWN ARROW)
 83A/     M     (DOWN ARROW)
 83B/           (DOWN ARROW)
 83C/     B     (DOWN ARROW)
 83D/     A     (DOWN ARROW)
```

The ASCII examination option displays every byte as its ASCII character. Those bytes which are not valid ASCII characters are displayed as blanks. The first two bytes of the EMPTAB are $01 and $03, not valid ASCII characters, and they displayed as blanks. The ZBUG A command lets you examine ASCII characters strings and messages easily.

It's kind of inconvenient not being able to display the numeric values, isn't it? Here's a trick in ZBUG. Use the = sign to **force** numeric display and Byte mode. To see the equivalent numeric, just enter an equals sign after each location, and you'll see the equivalent numeric:

```
#TABLE/         =01 (DOWN ARROW)
 838/           =03 (DOWN ARROW)
 839/     W     (DOWN ARROW)
 83A/     M     (DOWN ARROW)
 83C/           =20 (DOWN ARROW)
 83B/     B     (DOWN ARROW)
 83D/     A     (DOWN ARROW)
```

Accessing Multiple-Byte Table Entries

We can "scan" a table such as the one on the previous page just about as easily as we did in the one-byte per entry table case. Scanning means going through the table one entry at a time and trying to find a given entry.

In this case, though, we have to adjust the table pointers by 16 to get to the next entry.

Suppose that we are looking for the Social Security Number "key," held in memory locations $3000 and $3001. We can use the following program to scan the 4-entry EMPTAB table:

```
                 * TABLE LOOKUP FOR FINDING SOCIAL SECURITY NUMBER
ØB57 8E   ØB8Ø   SSNLOK  LDX   #EMPTAB LOAD START OF TABLE
ØB5A C6   Ø4             LDB   #4      FOR 4 ENTRIES
ØB5C B6   3ØØØ   SSNØ1Ø  LDA   $3ØØØ     GET FIRST BYTE OF #
ØB5F A1   84             CMPA  ,X        COMPARE
ØB61 26   ØB             BNE   SSNØ2Ø    GO IF NOT EQUAL
ØB63 3Ø   Ø1             LEAX  1,X       POINT TO NEXT BYTE
ØB65 B6   3ØØ1           LDA   $3ØØ1     GET 2ND BYTE OF #
ØB68 AØ   84             SUBA  ,X        COMPARE
ØB6A 27   ØC             BEQ   SSNØ25    GO IF "FOUND"
ØB6C 3Ø   1F             LEAX  -1,X      ADJUST PNTR
ØB6E 3Ø   88 1Ø  SSNØ2Ø  LEAX  16,X      POINT TO NEXT ENTRY
ØB71 5A                  DECB            # ENTRIES-1
ØB72 26   E8             BNE   SSNØ1Ø    GO IF NOT 4 ENTRIES
ØB74 86   FF             LDA   #$FF    "NOT FOUND" FLAG
ØB76 2Ø   Ø2             BRA   SSNØ3Ø
ØB78 3Ø   1F     SSNØ25  LEAX  -1,X    ADJUST X
ØB7A B7   3ØØ2   SSNØ3Ø  STA   $3ØØ2   STORE FLAG
ØB7D 7E   ØB7D   LOOP    JMP   LOOP    LOOP HERE
                 * TABLE WITH LARGE ENTRIES AND SEVERAL FIELDS
ØB8Ø      Ø1Ø3   EMPTAB  FDB   $Ø1Ø3
ØB82      57             FCC   'WM BABBAGE     '
          4D
          2Ø
          42
          41
          42
          42
          41
          47
          45
          2Ø
          2Ø
          2Ø
          2Ø
ØB9Ø      1Ø12           FDB   $1Ø12
ØB92      47             FCC   'GEORGE BOOLE   '
          45
          4F
          52
          47
          45
          2Ø
          42
```

Figure 11-6. Table Lookup Program 2

```
           4F
           4F
           4C
           45
           20
           20
0BA0       0F02       00340          FDB      $0F02
0BA2       42         00350          FCC      'BLAISE PASCAL '
           4C
           41
           49
           53
           45
           20
           50
           41
           53
           43
           41
           4C
           20
0BB0       101A       00360          FDB      $101A
0BB2       41         00370          FCC      'ALAN TURING   '
           4C
           41
           4E
           20
           54
           55
           52
           49
           4E
           47
           20
           20
           20
           0000       00380          END
00000 TOTAL ERRORS
           .
EMPTAB   0B80
LOOP     0B7D
SSN010   0B5C
SSN020   0B6E
SSN025   0B78
SSN030   0B7A
SSNLOK   0B57
```

Figure 11-6 continued

This is a fairly complicated program, so we'll explain carefully how it works. The flowchart is shown in Figure 11-7.

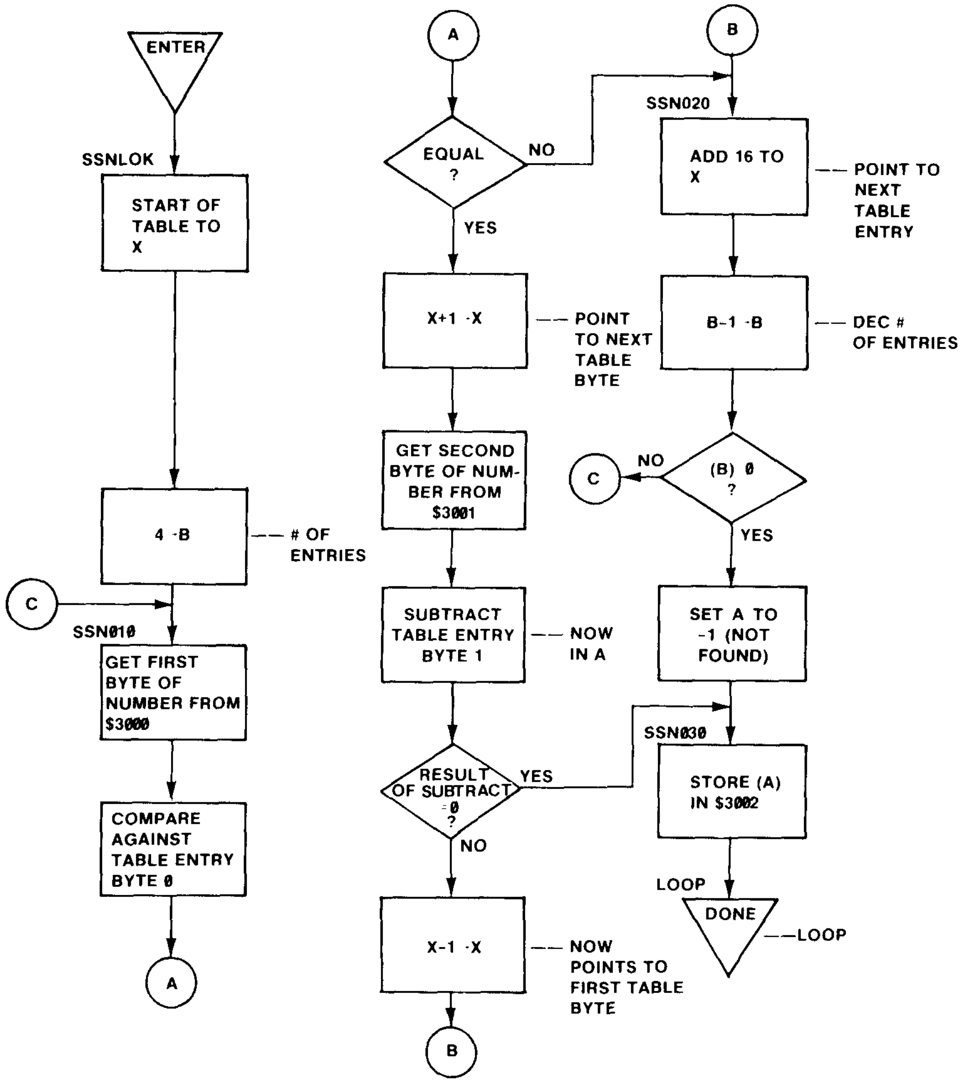


Figure 11-7. Table Lookup Program Flowchart

First of all, X is loaded with the address of the table. X will point to each entry in the table in turn.

Next, B is loaded with 4. Since there are 4 entries, we'll have to go through a loop 4 times to compare each entry.

The loop starts at SSN010. Each time through the loop, the following actions occur:

- The first byte of the Social Security number is loaded into A from memory location $3000.

- This value is compared to the table value pointed to by X. This would be the first byte of the "current" entry.
- If they are not equal (BNE), a jump is made to SSN020 to adjust the X pointer to the next entry.
- If they are equal, X is incremented by one by the LEAX to point to the next byte in the table entry.
- The next byte of the number in location $3001 is loaded into A.
- A subtract of ,X is done. This subtracts the next byte of the table entry from A, puts the result (zero or non-zero) in A, and sets the Condition Codes.
- If the second bytes were equal (BEQ), the number has been found. In this case a jump is made to SSN025.
- X is adjusted back to the start of the table entry by decrementing by one in the LEAX.
- If the first or second bytes were not equal, the LEAX 16,X adjusts X by 16 to point to the next table entry. The count in B is then decremented by one. A BNE then loops back to SSN010 for the next comparison. If the count in B is decremented down to 0 by the DECB, the loop is done, A is loaded with -1, and a jump is made to SSN030.
- SSN030 stores the value in A in location $3002. This value is 0 (from the subtract) if the number was found, or -1 if the number wasn't found after 4 compares.
- An important note: If the number is found, **X points to the table entry for the number.** This is the most important result of this table "scan."

If you care to run the program to see how it works, enter a number into $3000 and $3001 to correspond to one of the table entries. Breakpoint at LOOP and execute from SSNLOK.

At the end of the program, A should contain the "found"/"not found" flag and X should point to the table entry if it was found. Use the ZBUG R command to look at the registers.

We'll look at more table techniques in the next chapter.

Hints and Kinks 11-2
Using ZBUG U Command

Move data from one area to another by the ZBUG U command. To move 100 bytes from a table at location $2000 to a new area of $3000 do:

```
#U2000 3000 64
```

Note that the number of bytes is in hex.

Hints and Kinks 11-3
Using the ZBUG P, L, and V Commands

This is probably a good point to mention the ZBUG cassette tape commands. They let you save and load a "memory image" as a cassette tape file. This is handy for debugging — you can debug a file and then "checkpoint" the "object" to cassette. However, these three commands would be much **more** valuable on an Edit/Assembler package that was less interactive.

To dump any set of memory locations do a

```
PNAME NNNN MMMM EEEE
```

where NAME is the file name, NNNN is the start of the data to be dumped, MMMM is the end of the data to be dumped, and EEEE is the execution address. (If you're dumping data, use a dummy execution address; you won't be doing an execute anyway.)

V and L work the same as in the Edit mode.

Review

To review what we've learned in this chapter:

- A simple table might be a list of one-byte entries
- A table can be accessed by using an "index value." The index corresponds to one parameter, and the entry at the index position is another related value
- The FCB generates a one-byte data value
- Data can be labeled with symbolic names, just as instructions can be labeled
- Tables with multiple-byte entries are common
- Entries in tables may be subdivided into fields
- Tables may have "fixed-length" or "variable-length" entries
- The FDB generates two bytes of data
- The FCC generates a string of ASCII characters
- "Scanning" a table means that a search of the table is performed; the search is for a specific entry

For Further Study

FCB, FDB (EDTASM+ manual)
FCC and ASCII codes (EDTASM+ manual)

KEY CHART — CHAPTER 12

INSTRUCTIONS

~~X~~	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
~~CA~~	~~BITA~~	CLR	~~LDU~~	RORA
~~CB~~	~~BITB~~	~~CMPA~~	~~LDX~~	RORB
~~DA~~	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
~~DB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	RTI
~~DD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	RTS
~~DA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~DB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~DCC~~	~~LBLS~~	~~CMPY~~	LSLA	SEX
LA	~~BLT~~	~~COMA~~	LSLB	~~STA~~
LB	~~LBLT~~	~~COMB~~	LSL	~~STB~~
L	~~BMI~~	~~COM~~	LSRA	~~STD~~
RA	~~LBMI~~	CWAI	LSRB	~~STS~~
RB	~~BNE~~	DAA	LSR	~~STU~~
R	~~LBNE~~	~~DECA~~	MUL	~~STX~~
~~C~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~CC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~S~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~CS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~Q~~	~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~EQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~E~~	BSR	~~INCB~~	~~ORCC~~	SWI3
~~GE~~	LBSR	~~INC~~	PSHS	SYNC
~~T~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~GT~~	~~LBVC~~	JSR	PULS	~~TSTA~~
~~I~~	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~HI~~	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
~~S~~	~~CLRA~~	~~LDD~~	ROLB	

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

ADDRESSING MODES

~~HERENT~~
~~RECT~~
~~TENDED~~
~~MEDIATE~~
~~MPLE INDEXED~~
~~LATIVE~~
SPLACEMENT INDEXED
TO INCREMENT/DECREMENT
IRECT
PHISTICATED

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
INDEXING WITH X,Y	
SORTING	

PSEUDO OPS

U	ORG
~~B~~	RMB
~~C~~	SET
~~B~~	~~SETDP~~

ld Type **Present Chapter**
gular Type Future Chapters
~~lic Type~~ *Past Chapters*

Chapter 12
Indexing Operations Using X and Y

The X and Y registers in the simplest addressing case can be used as "pointers" to point to the address at which data can be found. Their real application, however, is in full "indexing" where they point to a midpoint location and allow data on either side of that location to be easily referenced.

Let's take another look at the program from Chapter 11. A modified version is shown below:

```
                    ØØ1ØØ * TABLE LOOKUP FOR FINDING SOCIAL SECURITY NUMBER
ØAC5 8E    ØAE8     ØØ11Ø SSNLOK  LDX   #EMPTAB LOAD START OF TABLE
ØAC8 C6    Ø4       ØØ12Ø         LDB   #4      FOR 4 ENTRIES
ØACA 1Ø8E  3ØØØ     ØØ13Ø         LDY   #$3ØØØ  POINT TO #
ØACE A6    A4       ØØ14Ø SSNØ1Ø  LDA   ,Y        GET FIRST BYTE OF #
ØADØ A1    84       ØØ15Ø         CMPA  ,X        COMPARE
ØAD2 26    Ø6       ØØ16Ø         BNE   SSNØ2Ø    GO IF NOT EQUAL
ØAD4 A6    21       ØØ17Ø         LDA   +1,Y      GET 2ND BYTE OF #
ØAD6 AØ    Ø1       ØØ18Ø         SUBA  +1,X      SUBTRACT
ØAD8 27    Ø8       ØØ19Ø         BEQ   SSNØ3Ø    GO IF "FOUND"
ØADA 3Ø    88 1Ø    ØØ2ØØ SSNØ2Ø  LEAX  16,X      POINT TO NEXT ENTRY
ØADD 5A             ØØ21Ø         DECB            #ENTRIES-1
ØADE 26    EE       ØØ22Ø         BNE   SSNØ1Ø    GO IF NOT 4 ENTRIES
ØAEØ 86    FF       ØØ23Ø         LDA   #$FF    "NOT FOUND" FLAG
ØAE2 B7    3ØØ2     ØØ24Ø SSNØ3Ø  STA   $3ØØ2   STORE FLAG
ØAE5 7E    ØAE5     ØØ25Ø LOOP    JMP   LOOP
ØAE8       Ø1Ø3     ØØ26Ø EMPTAB  FDB   $Ø1Ø3
ØAEA       57       ØØ27Ø         FCC   'WM BABBAGE     '
           4D
           2Ø
           42
           41
           42
           42
           41
           47
           45
           2Ø
           2Ø
           2Ø
           2Ø
ØAF8       1Ø12     ØØ28Ø         FDB   $1Ø12
ØAFA       47       ØØ29Ø         FCC   'GEORGE BOOLE  '
           45
           4F
           52
           47
           45
           2Ø
           42
           4F
           4F
           4C
           45
           2Ø
           2Ø
ØBØ8       ØFØ2     ØØ3ØØ         FDB   $ØFØ2
ØBØA       42       ØØ31Ø         FCC   'BLAISE PASCAL '
           4C
           41
           49
           53
           45
           2Ø
```

Figure 12-1. Table Lookup Program

```
          50
          41
          53
          43
          41
          4C
          20
0B18      101A      00320          FDB      $101A
0B1A      41        00330          FCC      'ALAN TURING    '
          4C
          41
          4E
          20
          54
          55
          52
          49
          4E
          47
          20
          20
          20
          0000      00340          END
00000 TOTAL ERRORS

EMPTAB   0AEB
LOOP     0AE5
SSN010   0ACE
SSN020   0ADA
SSN030   0AE2
SSNLOK   0AC5
```

Figure 12-1 continued

Compare this program with the version from Chapter 11 and see if you can find the differences.

Aside from the fact that the Y index register is used to point to the number to be found, the biggest difference is that X and Y are used not just as "pointers" but with a "displacement value" of +1 as in +1,X or +1,Y.

Indexed Addressing

The type of addressing mode we're using here is called "indexed addressing." We've used the X and Y registers previously in this book, but only as "register pointers" to point to a memory location. In this example, X and Y are used as "base index registers," pointing to the start of an area. The "+1" in the +1,X or +1,Y is a "displacement" value that is added to the X or Y pointer value to find the actual location pointed to. This location is called the "effective address."

Figure 12-2 shows what we mean. Here X points to location $3000. Doing an LDA in indexed addressing mode, however, allows us to load A not only with the contents of location $3000, but any memory byte in the addressing range of the 6809 — 65,536 locations in all (although some of them are not used as memory in the Color Computer).

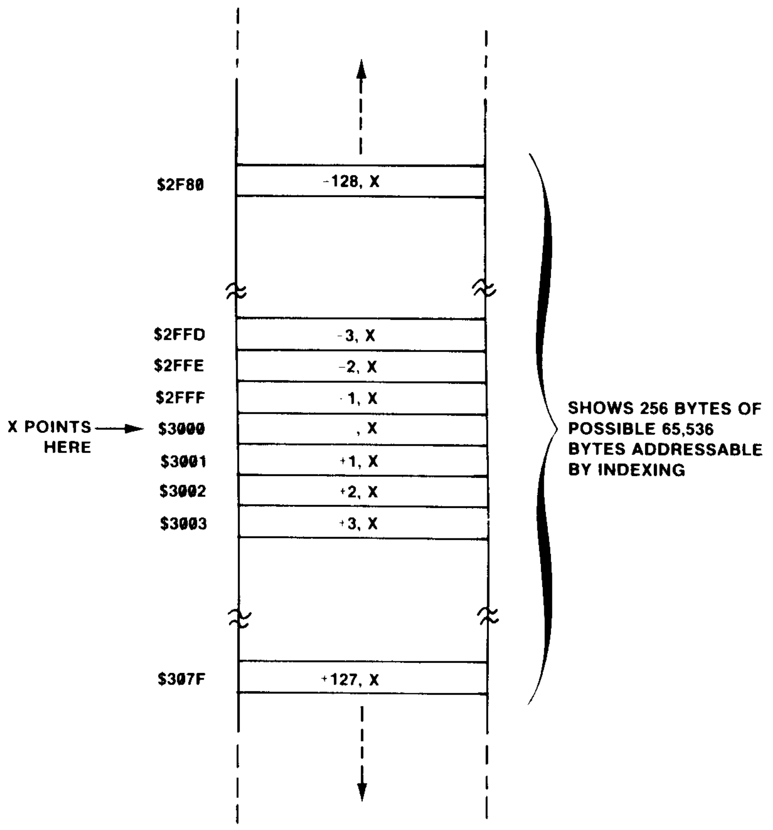


Figure 12-2. Indexed Addressing Example

Consider this program as an example:

```
                         00100 * INDEXING EXAMPLE
0952 8E    3003          00110 INDSTR  LDX     #$3003    POINT TO BASE
0955 A6    1D            00120         LDA     -3,X      GET -3 BYTE
0957 E6    03            00130         LDB     +3,X      GET +3 BYTE
0959 A7    03            00140         STA     +3,X      SWAP
095B E7    1D            00150         STB     -3,X
095D A6    1E            00160         LDA     -2,X      GET -2 BYTE
095F E6    02            00170         LDB     +2,X      GET +2 BYTE
0961 A7    02            00180         STA     +2,X      SWAP
0963 E7    1E            00190         STB     -2,X
0965 A6    1F            00200         LDA     -1,X      GET -1 BYTE
0967 E6    01            00210         LDB     +1,X      GET +1 BYTE
0969 A7    01            00220         STA     +1,X      LOOP
096B E7    1F            00230         STB     -1,X
096D 7E    096D          00240 LOOP    JMP     LOOP      LOOP HERE
           0000          00250         END
00000 TOTAL ERRORS

INDSTR  0952
LOOP    096D
```

Figure 12-3. Indexing Example Program

This isn't an especially profound program, but it does illustrate how indexing works. The data in $3000 through $3002 will be put into locations $3006 through $3004, respectively, by indexing the $3000 area. Note that X never changes. It points to the base address of $3003.

Now look at the machine-language instructions assembled for the indexed instructions. The first byte of indexed instructions are the "opcode." The last 1 to 3 bytes is always a "displacement" or a displacement code.

The displacement is a two's complement number. In the case of the instruction LDA -100,X, for example, the displacement is in the range of -128 through +127, very similar to the displacement used in the relative jump instructions. Assemble an LDA -100,X and look at the displacement in the third byte. The displacement byte here is $9C, which is a binary 10011100, or a -100 in two's complement form.

The "effective address" for an indexed instruction is found by taking the contents of the index register and adding the **sign extended** displacement to it. The result is the address used in the instruction.

Suppose we had $3003 in X.

In the case of LDA -100,X, for example, the effective address would be

```
00110000  00000000  ($3003 in X)
11111111  10011100  (-100 in displ)
------------------------------------
00101111  10011100  ($2F9C = EA)
```

"EA" stands for "effective address" in this computation. This effective address of $2F9C would be used in the LDA, so in effect:

```
LDA -100,X = LDA $2F9C
```

in this case.

If you look at the other indexed type instructions, you can see that they also have displacement bytes that duplicate the value in the source line. In the case of a displacement value from -16 to +15, the displacement value will usually be in the second byte of the instruction. It will be the last 5 bits of the byte. For displacements in the range -128 to -17 and +16 to +127, the displacement will usually be in the third byte of the instruction. For displacements of -32768 to -129 and +128 to +32767, the displacement will usually be in the third and fourth bytes of the instruction.

Of course, we're only showing you how the effective address is computed here for reference. You don't have to worry about the actual computation; the assembler will automatically take care of it for you. All you have to do is to establish the X or Y register at some base value and then use the proper displacement such as +23,X, -67,Y, +1000,X, or -300,Y.

Variable Offsets From X or Y

The 6809 is so versatile in indexing that it allows the displacement to be as small or as large as you wish, within the range of 65,536 bytes, of course.

Take these instructions:

```
LDY   #$3000     SETUP INDEX
LDA   ,Y         GET ($3000)
LDA   +5,Y       GET ($3005)
LDA   +64,Y      GET ($3040)
LDA   +$1000,Y   GET ($4000)
```

Just for fun, assemble them with an A/IM and look at the object code from the assembly. You'll see something like Figure 12-4.

```
0887 108E 3000      00100        LDY     #$3000   SETUP INDEX
088B A6   A4        00110        LDA     ,Y       GET ($3000)
088D A6   25        00120        LDA     +5,Y     GET ($3005)
088F A6   A8 40     00130        LDA     +64,Y    GET ($3040)
0892 A6   A9 1000   00140        LDA     +$1000,Y GET ($4000)
          0000      00150        END
00000 TOTAL ERRORS
```

Figure 12-4. Variable Offsets From X or Y

The first displacement for LDA ,Y was 0, and this resulted in a two-byte instruction with an opcode of $A6 and $A4 (no displacement).

The second displacement for LDA +5,Y was found in the last 5 bits of the second byte of $A6, $25.

The third displacement for LDA +64,Y was +64. This can be held in one byte, and this instruction was assembled with two opcode bytes of $A6 and $A8 for the opcode, followed by a third byte for the displacement, $40.

The fourth displacement for LDA +$1000,Y was hex 1000. This value cannot be held in 8 bits, but can be held in 16 bits, or 2 bytes. The assembler generated a $A6 and $A9 for the opcodes for this instruction, followed by a two-byte displacement of $10 and $00.

How does EDTASM+ know when to use no displacement byte, one displacement byte, or two displacement bytes? It's smart enough to figure out how large the displacement will be and generates a displacement accordingly. What this means is that you, as a programmer, can use any size displacement in indexed instructions and not have to worry about whether EDTASM+ can handle it!

Hints and Kinks 12-1
Indexing Formats

You can see the indexing formats in Appendix II under "Indexing Addressing Modes" and "Constant Offset From R." The indexing we're talking about in this chapter involves the X and Y registers, although in fact X, Y, and S and U can be used in the same fashion. We'll discuss the use of S and U in a later chapter.

The format of the indexed instruction depends upon the size of the displacement, the "offset."

If there is no offset, as in ,X, the the instruction consists of one or two opcode bytes, plus a last byte of 1RR00100 in binary, where "RR" is 00 for X or 01 for Y.

If the displacement can be held in 5 bits (-16 to +15, then the instruction consists of one or two opcode bytes plus a last byte of 0RRnnnnn, where RR is 00 for X or 01 for Y and nnnnn is the displacement value, in signed format.

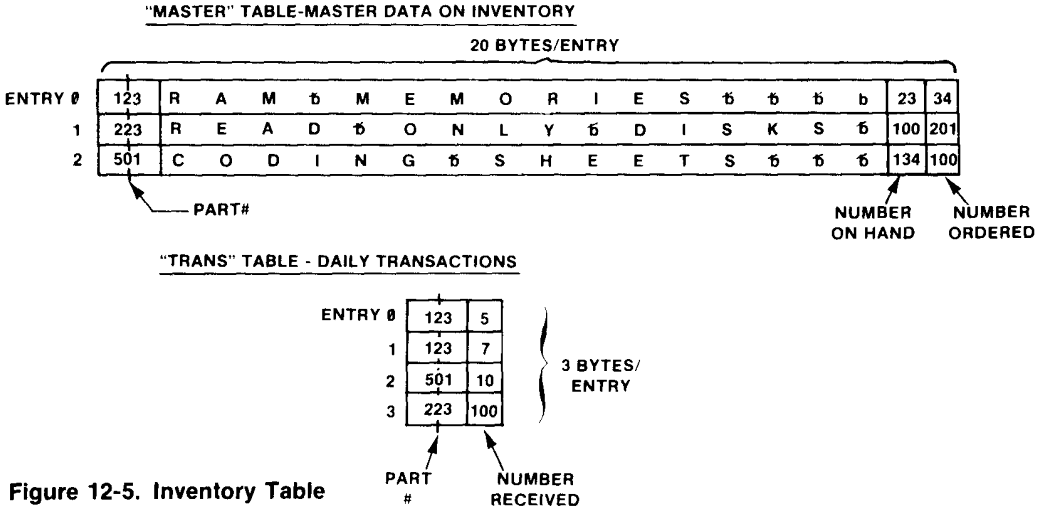
If the displacement can be held in 8 bits (-128 to +127), then the instruction consists of one or two opcode bytes, a byte of 1RR01000, where RR=00 for X and 01 for Y, and a displacement byte in two's complement.

If the displacement must be held in 16 bits, then the instruction consists of one or two opcode bytes, a byte of 1RR01001, where RR=00 for X and 01 for Y, and a 2-byte displacement value.

It's not necessary to know these formats bit for bit in programming the 6809, but we're presenting them here for your enlightenment and amusement!

Table Operations Using Indexing

Indexing is especially useful in working with tables. Suppose that we have a table that contains entries for a simple inventory system for a computer manufacturer. The table and entries are shown in Figure 12-5.



ENTRY 0	123	R	A	M	ƀ	M	E	M	O	R	I	E	S	ƀ	ƀ	ƀ	b	23	34
1	223	R	E	A	D	ƀ	O	N	L	Y	ƀ	D	I	S	K	S	ƀ	100	201
2	501	C	O	D	I	N	G	ƀ	S	H	E	E	T	S	ƀ	ƀ	ƀ	134	100

ENTRY 0	123	5
1	123	7
2	501	10
3	223	100

Figure 12-5. Inventory Table

Each entry in the "master" table is made up of four fields.

Field number 1 is the part number, from 0000 through 9999, in 2 bytes.

Field number 2 is the description of the part. This is a fixed-length field of 16 characters.

Field number 3 is the number "on hand," the number actually at the manufacturer, in one byte.

Field number 4 is the number on order, the number of parts that haven't yet come in, in one byte.

Each entry in the table is therefore 20 bytes long.

Each entry in the "transaction" table, a second table, is made up of two fields, a part number (two bytes) and the number received, in 1 byte. Each entry is therefore 3 bytes long.

We want to write a program that will adjust both the number on order and number on hand in the "master table" to reflect the number of parts that have come in from the "transaction" table. One way to implement it is shown below.

```
                        * INVENTORY PROGRAM USING INDEXING
                        * UPDATES MASTER TABLE WITH DATA FROM TRANSACTION
TABLE
0D15 108E 0D86          INVSTR  LDY     #TRANS   ADDRESS OF TRANSACTION
0D19 C6   04                    LDB     #4       NUMBER OF TRANSACTIONS
0D1B F7   0D49                  STB     COUNT    INITIALIZE COUNT
0D1E 8E   0D4A          INV010  LDX     #MASTER    ADDRESS OF MASTER
0D21 EC   A4            INV020  LDD     ,Y           GET PART #
0D23 10A3 84                    CMPD    ,X           TEST FOR PART #
0D26 26   12                    BNE     INV030       GO IF NOT EQUAL
0D28 A6   22                    LDA     +2,Y         GET NUMBER RECEIVED
0D2A AB   88 12                 ADDA    +18,X        ADD ON HAND, RECVD
0D2D A7   88 12                 STA     +18,X        STORE NEW ON HAND
0D30 A6   88 13                 LDA     +19,X        GET # ORDERED
0D33 A0   22                    SUBA    +2,Y         # ORDERED - # RCVD
0D35 A7   88 13                 STA     +19,X        STORE
0D38 20   05                    BRA     INV040       GO FOR NEXT PART
0D3A 30   88 14         INV030  LEAX    +20,X        POINT TO NEXT
0D3D 20   E2                    BRA     INV020       GET NEXT FROM MASTER
0D3F 31   23            INV040  LEAY    +3,Y       FOR TRANS TABLE
0D41 7A   0D49                  DEC     COUNT      DECREMENT #
0D44 26   D8                    BNE     INV010     GO IF NOT DONE
0D46 7E   0D46          LOOP    JMP     LOOP     DONE
0D49      00            COUNT   FCB     0        # COUNT
0D4A      007B          MASTER  FDB     123      PART NUMBER 123
0D4C      52                    FCC     'RAM MEMORIES    '
          41
          4D
          20
          4D
          45
          4D
          4F
          52
          49
          45
          53
          20
          20
```

Figure 12-6. Inventory Program

```
         20
         20
0D5C     17        00350          FCB     23
0D5D     22        00360          FCB     34
0D5E     00DF      00370          FDB     223       PART NUMBER 223
0D60     52        00380          FCC     'READ ONLY DISKS '
         45
         41
         44
         20
         4F
         4E
         4C
         59
         20
         44
         49
         53
         4B
         53
         20
0D70     64        00390          FCB     100
0D71     C9        00400          FCB     201
0D72     01F5      00410          FDB     501       PART NUMBER 501
0D74     43        00420          FCC     'CODING SHEETS    '
         4F
         44
         49
         4E
         47
         20
         53
         48
         45
         45
         54
         53
         20
         20
         20
0D84     86        00430          FCB     134
0D85     64        00440          FCB     100
0D86     007B      00450  TRANS   FDB     123       PART NUMBER 123
0D88     05        00460          FCB     5         5 RECEIVED
0D89     007B      00470          FDB     123       PART NUMBER 123
0D8B     07        00480          FCB     7         7 RECEIVED
0D8C     01F5      00490          FDB     501       PART NUMBER 501
0D8E     0A        00500          FCB     10        10 RECEIVED
0D8F     00DF      00510          FDB     223       PART NUMBER 223
0D91     64        00520          FCB     100       100 RECEIVED
         0000      00530          END
00000 TOTAL ERRORS

COUNT    0D49
INV010   0D1E
INV020   0D21
INV030   0D3A
INV040   0D3F
INVSTR   0D15
LOOP     0D46
MASTER   0D4A
TRANS    0D86
```

Figure 12-6 continued

The two tables are established in the program itself, one called MASTER and the other TRANS.

A flowchart for the program is shown in Figure 12-7.

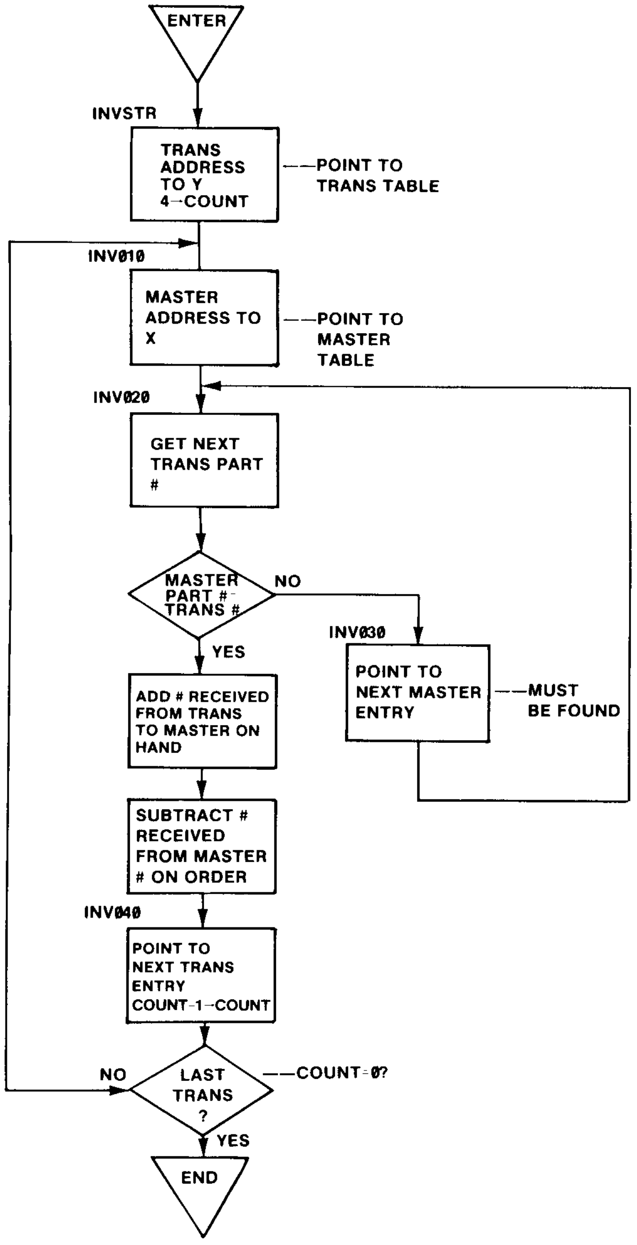


Figure 12-7. Inventory Program Flowchart

X is used in the program to point to the MASTER table. Y is used to point to the TRANS table. Y is used to go down the TRANS table one entry at a time. Each time one entry is obtained from TRANS, the MASTER table is "scanned" for the matching part number. **We're assuming the part number will always be found, by the way.** If it is not, what will happen?

Hints and Kinks 12-2
Inventory Program

No question about it, this program is flawed! If there is a TRANS part number that is not found in the MASTER table, the program keeps scanning the MASTER table for the part number. The instruction at INV030 continually increments the X register to point to the next part number in MASTER and doesn't stop with the last. The program as it stands will work fine, but is not good programming practice. We should have included a "terminating part number" in MASTER — something like a part number of 9999.

Maybe you'd like to "clean up" the program? Aha, I didn't think so. . .

When the part number is found, the number received from the TRANS table is added to the number on hand, and the updated number on hand is put back into the MASTER table. The number received is then subtracted from the number on order and the result is put back into the number on order.

All through the program, Y points to TRANS and X points to the master entries.

Here's a detailed description:

Y is first loaded with the address of TRANS. Next, B is loaded with 4. There are 4 entries in TRANS, so we'll have to make 4 updates to the MASTER table.

The "outer loop" starts at INV010. The outer loop is done 4 times, once for each entry in TRANS. The outer loop contains an "inner loop" which searches the MASTER table for the part number.

At INV010, the address of MASTER is loaded into X. In other words, we're starting again from the "top" of the MASTER table to look for the part number.

Next, the part number of the current TRANS entry is loaded into D (A and B). Notice the method in which D is loaded, by using Y indexing to load the next 2 bytes from TRANS.

The inner loop starts at INV020. At this point Y points to the current TRANS entry and X points to the first entry in MASTER.

D is now compared with the part number from the MASTER table. If the part numbers are not equal (BNE), INV030 is executed to add 20 to the X register. This points to the next entry in MASTER.

If the part numbers are equal, the number received is added to the number on hand, and the number received is subtracted from the number ordered. The results are stored back in MASTER. The outer loop code at INV040 is then executed. This code increments the Y (TRANS) pointer by 3 so that it points to the next TRANS entry. The number in B is then decremented and a branch is made back to INV010 for the next TRANS part number.

When all 4 TRANS part numbers have been processed, the program stops.

If you wish, you can assemble and run the program and you'll see the process. Use ZBUG to look at the results of the processing.

If you can follow the code in this program, you're doing very well. We have covered a great deal of ground in the past chapters and this program incorporates most of the concepts we've discussed in the past chapters. A great deal of 6809 assembly-language "code" is no more difficult than the program above.

Review

To recap what we've learned here:

- X and Y can be used as pointers, but may also be used as "index registers"
- When used as index registers, X and Y use a 5-bit, 8-bit, or 16-bit displacement value
- The displacement value, when "sign-extended" and added to the contents of the index register, forms an "effective address" that points to the operand for the indexed instruction
- Data can be referenced anywhere within memory from the index "base"
- Indexing can be used to an advantage in table and other operations

For Further Study

Instructions using X or Y indexing (Appendix II)

KEY CHART — CHAPTER 13

INSTRUCTIONS

~~ABX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
~~ADCA~~	~~BITA~~	**CLR**	~~LDU~~	RORA
~~ADCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	RORB
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
~~ADDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	RTI
~~ADDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	RTS
~~ANDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~ANDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ANDCC~~	~~LBLS~~	~~CMPY~~	LSLA	SEX
ASLA	~~BLT~~	~~COMA~~	LSLB	*STA*
ASLB	~~LBLT~~	~~COMB~~	LSL	~~STB~~
ASL	~~BMI~~	~~COM~~	LSRA	~~STD~~
ASRA	~~LBMI~~	CWAI	LSRB	~~STS~~
ASRB	~~BNE~~	DAA	LSR	~~STU~~
ASR	~~LBNE~~	~~DECA~~	MUL	~~STX~~
~~BCC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BCS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~LBEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~BGE~~	BSR	~~INCB~~	~~ORCC~~	SWI3
~~LBGE~~	LBSR	~~INC~~	PSHS	SYNC
~~BGT~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~LBGT~~	~~LBVC~~	JSR	PULS	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
~~DISPLACEMENT INDEXED~~
AUTO INCREMENT/DECREMENT
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
~~FCB~~	RMB
~~FCC~~	SET
~~FDB~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYT~~
~~P SAVE ML ON TAPE~~	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	SUBROUTINES
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
SORTING	

Bold Type – Present Chapter
Regular Type – Future Chapters
~~*Italic Type*~~ – *Past Chapters*

Chapter 13
Operations of a Different Sort and Unsigned Comparisons

In previous chapters, we've covered many of the programming techniques that we can use to access data in tables; using the indexed addressing mode plays an important part.

The tables before this chapter were "scanned" one entry at a time to find data; we didn't know exactly where in the table we'd find the "search key," and had to methodically go through all the entries until we found the one we were looking for. These types of tables are called "unordered," because the entries don't have any logical order.

In this chapter we'll look at tables with an "order." We'll also see how the index registers may be automatically incremented or decremented.

Types of Orders

In ordered tables, the order may be in alphabetic order, like a phone book, or in numerical order, historical order, or other scheme. Also, the order may be "ascending" or "descending." Descending would be a phone book printed from Z to A.

The most common order in assembly-language tables is alphabetic order, as in other types of programming. Usually the sequence is "ascending," from A to Z.

Actually, the "alphabetic" order really includes all ASCII characters, so it should really be called "alphanumeric" and special characters. Look at Appendix VI to see the ASCII codes for alphabetic, numeric, and special characters. The order we'll be using here will be based on these codes.

An example of data sorted on these codes is shown in Figure 13-1. Note that as you would expect, A through Z and 0 through 9 are kept in order, that blanks come before anything else, and that special characters are somewhat scattered around.

Hints and Kinks 13-1
More on Alphabetized Sorting

Although the Color Computer doesn't have lower-case (at this time of writing), lower-case characters are of higher weight than upper case, and would be **after** the same upper-case characters — "BARDEN" would appear **before** "aardvarke" in a sort of both upper- and lower-case characters!

A	A	R	D	V	A	R	K	ƀ	B	I	L	L	ƀ	ƀ	ƀ	
A	A	R	D	V	A	R	K	,	A	N	T	H	O	N	Y	
A	A	R	D	V	A	R	K	,	a	n	t	h	o	n	y	(lower case last)
A	A	R	D	V	A	R	K	—	a	n	t	h	o	n	y	(—after comma)
A	B	L	E		J	A	M	E	S	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	
A	B	L	E	!	J	A	M	E	S	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	(!after blank)
A	B	L	E	,	J	A	M	E	S	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	(comma after!)
A	B	L	E	,	C	H	A	R	L	I	E	ƀ	ƀ	ƀ	ƀ	
A	B	L	E	,	C	H	A	R	L	I	E	ƀ	ƀ	ƀ	ƀ	(h after H)
A	B	L	E	,	J	O	H	N	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	
A	B	L	E	,	J	O	H	N	ƀ	C	•	ƀ	ƀ	ƀ	ƀ	
A	B	L	E	,	J	O	H	N	ƀ	C	A	R	T	E	R	(period before C)

ƀ=BLANK

Figure 13-1. Sorted Data

Sorting

How do tables get in order? Suppose that we're entering a list of names as we think of them. How do we order them? This process is called "sorting" and it means that we're ordering the data according to some scheme, usually alphanumeric.

Because sorting long tables or lists of data involves a great deal of processing and sorting is such a common technique, there are many different sort schemes — the bubble sort, the two-buffer sort, the Shell and Shell-Metzner sorts, and others. You might want to look at some of these other schemes in detail. You can find the "algorithms" in computing magazines.

One of the most common sorts used in assembly language is the "bubble sort." The bubble sort operates as shown in Figure 13-2.

Hints and Kinks 13-2
Bubble Sorts

Bubble sorts have two outstanding characteristics: 1. They are horrendously slow. 2. They use no extra space in the sort process.

Because we're using 6809 assembly language, the bubble sort speed will be acceptable, and we can see the whole process on the screen as we're using only one sort "buffer."

You might want to look into some other sort processes. There has been a great deal written on efficient sorts — books and books worth of material. If dueling were still in vogue, there'd be dozens of dead computer scientists, each having perished in defense of his sort.

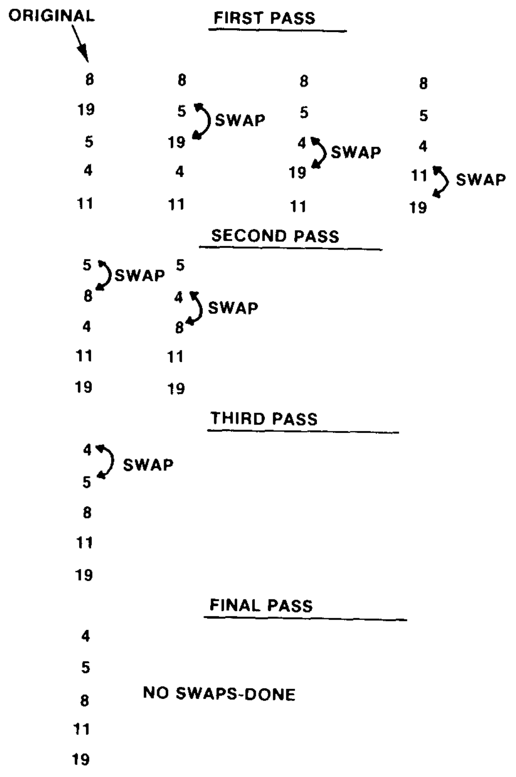


Figure 13-2. Bubble Sort Operation

A table of entries is originally "unordered."

The sort starts with the first two entries and compares them. If the second entry is of lower "order" than the first, the two entries are "swapped." If the second entry is equal to or greater than the first, no swap is made.

The second and third entries are now compared, and a swap is made or the entries are left unchanged. This process continues until the last two entries in the table are compared.

At the end of the first pass, the table entries are usually not ordered. Other passes that repeat the compare and swap process have to be made. The swapping continues until the entries are in alphanumeric order.

As the "lighter" entries "bubble" to the top, this sort is called a bubble sort.

We've programmed a bubble sort on the top of the next page:

```
                         00100 * BUBBLE SORT
0A6A 7F    0A95          00110 BUBSRT   CLR     PASSNO   SET PASS # TO 0
0A6D 8E    0400          00120 BUB010   LDX     #$400       POINT TO SCREEN
0A70 108E  0000          00130          LDY     #0          SET CHANGE FLAG TO 0
0A74 A6    80            00140 BUB020   LDA     ,X+            GET FIRST ENTRY
0A76 A1    84            00150          CMPA    ,X             TEST NEXT
0A78 23    0A            00160          BLS     BUB030         GO IF A<=B
0A7A E6    84            00170          LDB     ,X             GET SECOND ENTRY
0A7C E7    1F            00180          STB     -1,X           SWAP B TO A
0A7E A7    84            00190          STA     ,X             SWAP A TO B
0A80 108E  0001          00200          LDY     #1             SET "CHANGE"
0A84 8C    05FF          00210 BUB030   CMPX    #$400+511      TEST FOR SCREEN END
0A87 26    EB            00220          BNE     BUB020         GO IF NOT ONE PASS
0A89 7C    0A95          00230          INC     PASSNO      INCREMENT PASS #
0A8C 108C  0000          00240          CMPY    #0          TEST CHANGE FLAG
0A90 26    DB            00250          BNE     BUB010      GO IF CHANGE OCCURRED
0A92 7E    0A92          00260 LOOP     JMP     LOOP     LOOP HERE
0A95       00            00270 PASSNO   FCB     0        PASS #
           0000          00280          END
00000 TOTAL ERRORS

BUB010   0A6D
BUB020   0A74
BUB030   0A84
BUBSRT   0A6A
LOOP     0A92
PASSNO   0A95
```

Figure 13-3. Bubble Sort Program

The program works with the text screen, which starts at RAM location $400 (see my Radio Shack book "Color Computer Graphics" for more information on graphics). The program assumes that this area is a table of data; each entry in the table is a one-byte entry, so there are a total of 32 bytes in the table.

The bubble sort sorts whatever is on the screen, putting the "lower" order bytes at the beginning of the screen.

You can use the text from the assembly as sample data, or fill the screen with ASCII characters using ZBUG. To see how it works, don't breakpoint, but execute from BUBSRT after assembly. You can press RESET (on the back of the Color Computer) to get back to EDTASM+.

If you do run the program, notice how the data has been ordered in alphanumeric order. The bubble sort is inherently slow, but is fun to watch.

Let's see how the program works. First of all, a warning — we're "flexing our programming muscles here" and starting to use some of the power of the 6809, so we've thrown in a couple of new concepts. If you've come this far, though, you shouldn't have any problem with them!

The inner loop goes from BUB020 through the BNE BUB020 and is used to "scan" down through the table from beginning to end. At BUB020, X points to the start of the table at $400. The Y register is set to 0 at the beginning of each new inner loop pass. If any swap occurs, then it is set to 1. At the end of the pass, if Y is still set to 0, no swap has occurred and the table is sorted.

The outer loop starts at BUB010 and goes to the end of the program. This loop initializes X and Y for each new pass at the beginning of the pass. At the end of the pass, it increments the pass number in "variable" PASSNO, and then tests the change flag in Y. If a change occurred, the table is still not sorted, and another pass is made by looping back to BUB010.

Hints and Kinks 13-3
CLR to Memory

I **hate** loose ends. Early on in this book we talked about CLRA and CLRB, but didn't mention a CLR to memory. Yes, you can clear a memory location by using the CLR instruction in direct, extended, or indexed mode. It's a convenient instruction to have.

There are a couple of "tricky" points that haven't been covered up to this point. These points are:

- Using the "Auto Increment" feature of indexing
- Labeling a variable location
- Using expressions
- Using unsigned comparisons

"Auto" Incrementing and Decrementing

The first of these points is "auto" increment or decrement. Since the index registers are constantly changed by plus or minus one, why not make this an optional feature when using indexing? The 6809 lets you do this, by specifying a + or - after the index register as in the LDA ,X+ instruction above. However, you can not only increment or decrement by **one** count, you can increment or decrement by one or **two** counts. Here are some examples:

```
LDA   ,X+   (load A and then increment X)
LDA   ,Y++  (load A and then increment Y twice)
LDA   ,-Y   (decrement Y and then load A)
LDA   ,--X  (decrement X twice and then load A)
```

Note that the increment is done **after** the normal instruction operation and is the same as an "LEA" instruction following the indexed operation, and that the decrement is done **before** the normal instruction operation and is the same as an "LEA" instruction before the indexed operation.

The auto increment and decrement are extremely useful, and are used all the time.

Hints and Kinks 13-4
More on Auto Increment and Decrement

Auto increment and decrement and displacement (offset) indexing (as in LDA 30,X) are mutually exclusive, that is, you can't have an instruction such as LDA 30,X++.

Labeling a Variable Location

We used a RAM location here to store a count of the number of passes that were made through the bubble sort. We could have used an "absolute" location such as $3000, but chose instead to use a location close to the program. You can refer to any variable or constant by a symbolic name instead of an absolute value, and, in fact, this is the preferred way to use variable storage.

To examine "PASSNO" at the end of the bubble sort, write down the location of PASSNO from the symbol table listing at the end of the assembly and use the ZBUG slash command, or simply use

```
#PASSNO/
```

to examine the contents.

Using Expressions

You'll note that we used an **expression** in the source code above, #$400+511. This expression defines the last location of the text screen. When EDTASM+ encounters the expression it will easily compute the value $5FF and use that value as the immediate data value in the CMPX instruction. EDTASM+ is very capable of using not only expressions with adds and subtracts, but other "operators" as well. We'll go into more detail about this in following chapters.

Hints and Kinks 13-5
Expressions

There are a number of "operators" that can be used in both ZBUG and the assembler portion of EDTASM+. Along with + (addition or sign of number) and - (subtraction or sign of number), you can also multiply (*), divide (.DIV.), find the modulus (.MOD.), logical shift (<), AND (.AND.), OR (.OR.), exclusive OR (.XOR.), and complement (.NOT.). You can also use the two relational operators for equals (.EQU.) and not equal (.NEQ.). See the EDTASM+ manual for more details.

As a practical matter, the operators you'll be using most frequently in Color Computer assembly-language programming will be:

- Addition/subtraction, as in
```
TABLE FDB    $400+DISP-1
```
- Multiplication or division, as in
```
TABLE FDB    16*NENT
LOCA  FDB    SIZE/2
```
- Shifting, as in

```
TABLE FDB    13<8      SHIFT 8 BITS LEFT
      FDB    LOCA<-8   SHIFT 8 BITS RIGHT
```

The other operators are embellishments, and not used as frequently, except for special cases (or by my cousin, who happens to be a modulus freak).

Using Unsigned Comparisons

The CMP can be used to compare two bytes in **unsigned** fashion as follows: The compare compares A with another operand, call it B, by subtracting B from A: A-B.

If A is less than B, the Carry Condition Code will be set (CS). If A is equal to or greater than B, the Carry Condition Code will be reset (CC). This condition always applies for a CMP, or for an SUB.

You really don't have to remember all of this! Just use the proper mnemonic in the BR instruction as detailed in Chapter 8. We used "BLS" in the program above to branch if the A operand was "less than or the same" as the B operand. You must know, though, whether you're dealing with two's complement or unsigned (absolute) numbers when choosing mnemonics as described in Chapter 8.

A Bubble Sort of A Two-Byte Entry Table

We'll continue in this vein with the following problem: Suppose you had a table made up of two-byte entries at the text screen locations $400 through $5FF, as shown in Figure 13-4.

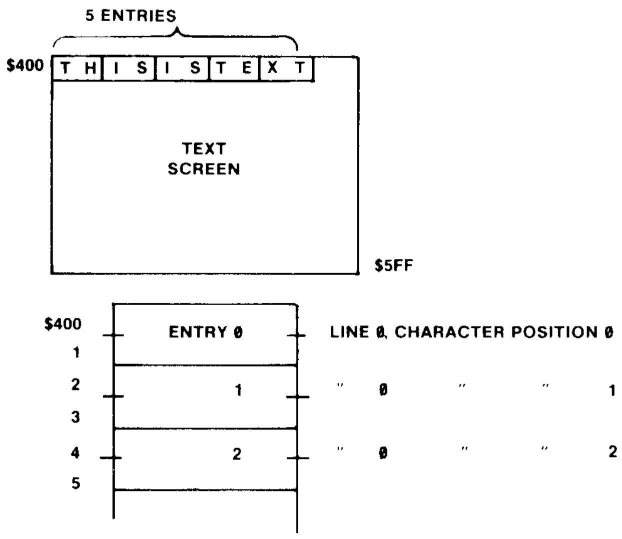


Figure 13-4. Two Byte Entry Table

How would you write a bubble sort to work with these two-byte entries?

Hints: You'll want to use the D register as the "accumulator" and compare to memory for the second operand. You could use either X or Y to point to the table entries, incementing the index register by two after each comparison. You'd also need to hold a "change" flag in a memory variable.

Here's the answer: There are many ways to do this, so if your ideas are different than this one, don't let it bother you. The one we've come up with is this:

```
                         00100 * BUBBLE SORT FOR TWO-BYTE TABLE
0A1E 8E   0400           00110 BUBSRT  LDX     #$400    POINT TO SCREEN
0A21 7F   0A45           00120         CLR     CHANGE   RESET CHANGE FLAG
0A24 EC   81             00130 BUB011  LDD     ,X++       GET FIRST ENTRY
0A26 10A3 84             00140         CMPD    ,X         COMPARE
0A29 23   0D             00150         BLS     BUB021     GO IF A<=B
0A2B 10AE 84             00160         LDY     ,X         GET 2ND ENTRY
0A2E ED   84             00170         STD     ,X         SWAP ENTRIES
0A30 10AF 1E             00180         STY     -2,X
0A33 86   01             00190         LDA     #1         ONE
0A35 B7   0A45           00200         STA     CHANGE     SET CHANGE FLAG
0A38 8C   05FE           00210 BUB021  CMPX    #$400+510 TEST FOR END
0A3B 26   E7             00220         BNE     BUB011     GO IF NOT ONE PASS
0A3D B6   0A45           00230         LDA     CHANGE   TEST CHANGE FLAG
0A40 26   DC             00240         BNE     BUBSRT   GO IF CHANGE OCCURRED
0A42 7E   0A42           00250 LOOP    JMP     LOOP     LOOP HERE
0A45      00             00260 CHANGE  FCB     0        INITIALLY 0
          0000           00270         END
00000 TOTAL ERRORS

BUB011  0A24
BUB021  0A38
BUBSRT  0A1E
CHANGE  0A45
LOOP    0A42
```

Figure 13-5. Bubble Sort for Two-Byte Table Program

Here we used an auto increment of "++" which increments the X register by 2 **after** the first entry is loaded. X now points to the next entry, 2 bytes following. Notice that the "swap" is done by using a displacement of -2, to point back to the original entry location.

Hints and Kinks 13-6
The Two-Byte Entry Table Sort

If you run this sort you'll see sorted data on the screen grouped in two bytes. A typical sequence on the first line might be "AAACAEA0A2A4A6A8BS. . .," corresponding to AA/AC/AE/A0/A2/A4/A6/A8/BS. . .

We'll leave it up to you to investigate this program in detail using ZBUG, except for one important point: What does the LDY ,X do and is it legitimate? The answer is that it loads the Y register in the same way D is loaded, and yes, it is proper to load the Y register in the indexed mode. As a matter of fact, X, Y, S, and U can all be used as index registers and can do **indexing**; we'll be investigating this topic in a later chapter.

Review

To review what we've learned in this chapter:

- Tables may be ordered in alphanumeric or other order
- The order may be "ascending" or "descending"
- Many tables are in alphanumeric and ascending order
- Various types of sorts are used to put tables in order
- The bubble sort works by comparing pairs of entries in a table and swapping if the first is of higher value than the second
- The Carry Condition Code can be used to compare two values in CMPs, SUBs, or SBCs
- The Carry Condition Code is set (CS) if the first operand is less than the second operand and reset (CC) if the first is equal to or greater than the second; normally you don't have to be aware of when the Carry is set because of BR "mnemonics"

For Further Study

Carry Condition Code setting for CMP, SUB, SBC (Appendix II)

KEY CHART — CHAPTER 14

INSTRUCTIONS

~~ABX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
~~ADCA~~	~~BITA~~	~~CLR~~	~~LDU~~	RORA
~~ADCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	RORB
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
~~ADDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	RTI
~~ADDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	**RTS**
~~ANDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~ANDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ANDCC~~	~~LBLS~~	~~CMPY~~	LSLA	~~SEX~~
ASLA	~~BLT~~	~~COMA~~	LSLB	~~STA~~
ASLB	~~LBLT~~	~~COMB~~	LSL	~~STB~~
ASL	~~BMI~~	~~COM~~	LSRA	~~STD~~
ASRA	~~LBMI~~	CWAI	LSRB	~~STS~~
ASRB	~~BNE~~	DAA	LSR	~~STU~~
ASR	~~LBNE~~	~~DECA~~	MUL	~~STX~~
~~BCC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BCS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~LBEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~BGE~~	**BSR**	~~INCB~~	~~ORCC~~	SWI3
~~LBGE~~	**LBSR**	~~INC~~	PSHS	SYNC
~~BGT~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~LBGT~~	~~LBVC~~	**JSR**	PULS	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	ROLB	

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	*M(OVE)*	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABL~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOC~~
C(ONTINUE)	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDIN~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BY~~
~~P SAVE ML ON TAPE~~	: FORCE FLAGS
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
~~DISPLACEMENT INDEXED~~
~~AUTO INCREMENT/DECREMENT~~
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
~~FCB~~	RMB
~~FCC~~	SET
~~FDB~~	~~SETDP~~

GENERAL TOPICS

~~CPU REGISTERS~~	**SUBROUTINES**
~~DATA TO REGISTERS~~	STACK OPERATIONS
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 14
Using Subroutines in Assembly Language

Subroutines are segments of code that exist in one place in memory. These sequences of code can be executed by temporarily transferring control to the code segment.

Subroutines may be from one to thousands of instructions long. If any set of instructions is executed more than one time, the instructions may be made into a subroutine in one place in memory to save memory space, and to save time in "coding."

The whole process is virtually identical to the BASIC GOSUB process. In this chapter we'll look at how subroutines are used and which instructions are involved.

Subroutine Basics

A subroutine is "called" by a "Branch to Subroutine" or BSR instruction; an RTS instruction marks the end of the subroutine. A simple "timing loop" subroutine, for example, might be something like:

```
                    00100 * DECREMENT X DOWN TO 0 AS TIMING LOOP
08DB 86   B1        00110 TIMELP  LDA     #177     COUNT IN A
08DD 4A             00120 TIM010  DECA               INNER LOOP
08DE 26   FD        00130         BNE     TIM010     GO IN NOT 0
08E0 30   1F        00140         LEAX    -1,X     OUTER LOOP
08E2 26   F7        00150         BNE     TIMELP   GO IF NOT 0
08E4 39             00160         RTS              RETURN
          0000      00170         END
00000 TOTAL ERRORS

TIM010  08DD
TIMELP  08DB
```

Figure 14-1. Simple Timing Loop

This timing loop might be called every time that you wanted a delay in the program; it delays about 1/1000 second (1 millisecond) for every count in X, to show you how fast assembly language is. A typical call would be

```
LDX   #200      LOAD TIMING COUNT
BSR   TIMELP    DELAY 200 MILLISECONDS
...             RETURN HERE
```

Hints and Kinks 14-1
The Timing Loop

Timing loops are used quite frequently to time such things as cassette tape and RS-232-C input/output drivers, any function which must have a correlation to "real time." Appendix II lists the cycles (wavy line) for each instruction and addressing type. To find the total time of execution of an instruction, multiply the cycle time by 1.124 microseconds (millionths of a second) on the Color Computer. You can construct your own timing loops by counting the times for each instruction and inserting the proper "padding" to round out a loop to a convenient unit of time, as we've done in the TIMELP program.

As you can see from the above, a return is made to the instruction after the BSR. The 6809 records the location of the next instruction after the call by taking the contents of the PC (Program Counter) and saving it in the "stack," an area of RAM. The RTS instruction retrieves the location from the stack and puts it back into the PC to cause a jump back to the location after the BSR.

Let's investigate what the stack is and where it is located to see how the BSR and RTS work. Look at this program:

```
093B 8E    03E8         START   LDX     #1000     DELAY COUNT
093E 8D    03                   BSR     TIMELP    CALL DELAY
0940 7E    0940         LOOP    JMP     LOOP      LOOP HERE
                        * DECREMENT X DOWN TO 0 AS TIMING LOOP
0943 86    B1           TIMELP  LDA     #177      COUNT IN A
0945 4A                 TIM010  DECA                INNER LOOP
0946 26    FD                   BNE     TIM010      GO IF NOT 0
0948 30    1F                   LEAX    -1,X      OUTER LOOP
094A 26    F7                   BNE     TIMELP    GO IF NOT 0
094C 39                         RTS               RETURN
           0000                 END
00000 TOTAL ERRORS

LOOP     0940
START    093B
TIM010   0945
TIMELP   0943
```

Figure 14-2. Timing Loop Program

If you assemble and execute the code above, you be should be able to see how the stack works. Breakpoint at HERE:

```
*A/IM
*Z
#XLOOP
#GTSTART
```

After the breakpoint is reached, use the R command to look at the 6809 registers. Look at the contents of the S register. The S register is the register which points to the current location of the stack. Take the address in the S register and subtract 4 from it, and then examine the locations at the stack area. You'll have something like this:

```
0 BRK @ LOOP
#R
 A=00 B=00 DP=00 CC=80 =E
 X=0000 Y=0000 U=0000 S=0777
 PC=084E
#B
#773/     0
```

```
774/   0
775/   8
776/   4F
777/   0
```

Here's what should happen when you execute the program: The S register pointed to a stack area in ZBUG.

The BSR jumped to location TIMELP. The jump action is the same as a BRA as far as the transfer of control.

The BSR, however, does one more thing. If you look at locations $775 and $776 (or the locations in your example, which will probably be different), you'll see the address of the return point after the BSR. The BSR has stored the return point by using the S register as a pointer.

Look at the contents of the S register. It is now 2 counts higher than where the return address was stored.

The S register is automatically decremented by 2 for every return address stored in the stack area. The "stack area" in this case started at $777 and "builds" down, as the BSR decrements the SP register by 2.

As soon as the subroutine (timing loop) is over, the return address is loaded into the PC from the stack as the RTS is executed. At the same time, the S register is **incremented** by 2. After the RET, the S again points to location $777, and the instruction after the BSR at LOOP is executed.

The S Stack

The operation above is pretty typical of how the stack is used for subroutine calls. The stack can be any area of memory that will be unused by the system or by your own program. Of course, it has to be RAM, as we're both reading and writing into it by BSRs and RTSs.

The stack area is normally set once at the beginning of the program, by the

```
        LDS        #$778
```

or similar instruction. Note that the S register is loaded with **one more** than the first stack location used. The S register is always **decremented first,** before the store of the BSR address.

Normally, you don't have to worry too much about setting the S register. BASIC, EDTASM+, and other programs **always** set the S to the stack area used in their programs, and it's available for everybody else's use too. Sometimes, though, you want to control the stack yourself, and in that case you'll use the LDS to point to your own stack area.

Normally the stack area should be about 100 bytes or so. All this means is that you must make certain that the S register is set to the last location of a memory area that won't be used by anything else.

Hints and Kinks 14-2
The S Stack

The S stack is the dedicated "hardware" stack of the 6809. It is specifically designed to hold the return point addresses for BSR, LBSR, and JSR subroutine "calls," along with other functions.

There is a second stack, the U(ser) stack in the 6809. The U stack is never used by the "hardware" to store return addresses, but is an optional stack which the user defines for storage of temporary data, indexing, or block operations.

Nested Stack Calls

Not only can you BSR a single subroutine, but that subroutine can BSR another subroutine, and that subroutine can BSR another, and so forth. How many BSRs can you make? Theoretically, as many as you want. Each time you make a BSR, another return address is saved in the stack, and that takes 2 bytes. If you have 10 "levels" of subroutines, you've used up 20 bytes of the stack, and the S register points to the original location minus 20 bytes.

Let's see an example of a "nested" set of subroutines:

```
* NESTED SUBROUTINES
NESTSR   BSR    SUBR1   CALL SUBROUTINE 1
LOOP     JMP    LOOP    RETURN POINT 1
SUBR1    BSR    SUBR2   CALL SUBROUTINE 2
         RTS            RETURN POINT 2
SUBR2    BSR    SUBR3   CALL SUBR 3
         RTS            RETURN POINT 3
SUBR3    BSR    SUBR4   CALL SUBR 4
         RTS            RETURN POINT 4
SUBR4    RTS            RETURN
         END
```

If you assemble and execute this program, breakpointing at LOOP, locate the stack area by looking at the contents of S, and then examine the stack area, you'll see something like this:

The BSR SUBR1 should have stored the address of the JMP LOOP instruction into the upper two stack locations.

SUBR1 consists of another BSR, to SUBR2. This should have stored the address of return point 2 into the next two stack locations.

SUBR2 consists of another BSR, to SUBR3. This should have stored the address of return point 3 into the next two stack locations.

SUBR3 consists of a fourth call, to SUBR4. This should have stored the address of return point 4 into the next two stack locations.

At this point the stack area appears as shown in Figure 14-3. The four return points are "4 levels deep" in the stack, and the S register points to a location 8 bytes down from the original value.

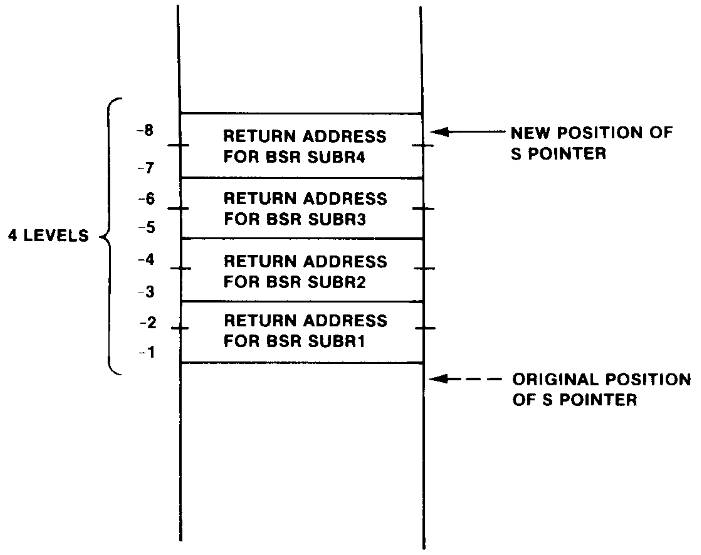

4 LEVELS
-8
-7
-6
-5
-4
-3
-2
-1
RETURN ADDRESS FOR BSR SUBR4
RETURN ADDRESS FOR BSR SUBR3
RETURN ADDRESS FOR BSR SUBR2
RETURN ADDRESS FOR BSR SUBR1
NEW POSITION OF S POINTER
ORIGINAL POSITION OF S POINTER

Figure 14-3. Stack Area Example

You'll be able to see all of the return addresses by looking at the stack area and comparing the addresses to the addresses in the assembly listing or the symbol table.

When the RTS in SUBR4 is executed, the RTS causes return point 4 to be put into the PC, causing a branch to return point 4. The stack pointer now points to a location 2 bytes higher.

Return point 4 is also an RTS, causing a branch to return point 3, and resetting the S register to a location 2 bytes higher.

Return point 3 is also an RTS, causing a branch to return point 2, and resetting the S register to a location 2 bytes higher.

Return point 2 is also an RTS, causing a branch to return point 1, and resetting the S register to its original setting.

This type of nesting is very common in many programs. Of course, this program does nothing except to illustrate the stack, and normal programs would have a great deal of "code" between the entry to the subroutine and the next BSR or RTS.

Usually you won't use more than about 3 or 4 levels of subroutines. It's just too hard to keep track of where you are if you use more, and usually not necessary.

A BSR for Every RTS

If you look at the program above, you'll see a RTS instruction executed for every BSR executed. If there wasn't an RTS for every BSR, what would happen?

The answer is that the stack would get "out of sync." The wrong return point would be picked up on a RTS. If you repetitively call the stack with this condition, you'll soon run out of stack as the stack "built down" and into another memory area, probably part of your program, or a system program. That will put "garbage" (in precise programming terms) into the program area and cause your program or the system program to blow up.

This condition is called stack overflow (or underflow, depending upon which direction the stacks goes out of bounds) and is a common programming bug!

For example, if you had:

```
EXAMP    BSR      SUBR1  BRANCH TO SUBR
         ...
SUBR1    ...             SUBROUTINE
         ...             CODE HERE
         ...
         ...
         BRA      EXAMP  SHOULD BE RTS!
```

the BRA back to EXAMP would cause SUBR1 to be called again, putting the return point in the next two bytes, and this would continue indefinitely with the return point being stored in lower and lower locations until the stack overflowed into a program area.

Hints and Kinks 14-3
An RTS for Every BSR?

Most of the time you will have an RTS for every BSR, especially in these simple programs. However, it's not really necessary to have an RTS for every call, as long as the S register is "reset." In lieu of an RTS for a single-level stack call, for example, you could do an LEAS

2,S which would add 2 to the S register, an action identical to the arithmetic action of an RTS. This concept could be expanded for any number of levels or stack usage.

Other Branch To Subroutines

The BSR instruction is a relative addressing instruction identical to the other BRs as far as the displacement field and "range." As with the other relative branches, the BSR has a "long branch" form so that a BSR can be done anywhere in memory without having to worry about being out of displacement range (128 locations back or 127 locations forward from the return point):

```
        LBSR    WAYOUT    LONG BRANCH TO SUBR
```

There's also a "Jump to Subroutine" or a JSR. Why have two forms of an unconditional branch to a subroutine? The JSR is a leftover from the 6800, the predecessor of the 6809. The JSR always assembles with an absolute address in the instruction rather than a "relative" displacement. For reasons that we'll get into in a later chapter, this type of instruction is not "position independent" or "relocatable." It's probably best to always use the LBSR rather than the JSR. Figure 14-4 shows an assembly of the two types and shows the format of each.

```
088B 17    0006     00110          LBSR    SUB1     LONG BRANCH BSR
088E BD    0894     00120          JSR     SUB1     EXTENDED ADDRESSING
0891 7E    0891     00130 LOOP     JMP     LOOP
0894 39             00140 SUB1     RTS              DUMMY SUBROUTINE
           0000     00150          END
00000 TOTAL ERRORS

LOOP     0891
SUB1     0894
```

Figure 14-4. JSR Vs. LBSR

Review

To review what we've learned in this chapter:

- Subroutines are a collection of instructions that are used more than once but are in one location in memory
- Subroutines save space and coding time
- Subroutines may be from 1 to many instructions long
- Subroutines are called by a BSR, LBSR, or JSR instruction and ended by an RTS instruction
- A BSR, LBSR, or JSR saves the address of the instruction after the BSR, LBSR, or JSR in the stack

- An RTS gets the last return address from the stack and causes a jump back to the return point
- The stack area in memory is any convenient memory area that can be set aside for stack actions
- The S (hardware stack pointer) points to the stack area
- The stack area "builds down"; each BSR, LBSR, or JSR stores 2 bytes of the return address into the next 2 lower stack locations
- Subroutines can be "nested" as often as required
- There must be an RTS executed for every BSR, LBSR, or JSR, or at least an adjustment of the S register pointer

For Further Study

Instruction formats for BSR, LBSR, JSR, and RTS (Appendix II)

KEY CHART — CHAPTER 15

INSTRUCTIONS

	~~LBHS~~	~~CLRB~~	~~LDS~~	ROL
	~~BITA~~	~~CLR~~	~~LDU~~	RORA
	~~BITB~~	~~CMPA~~	~~LDX~~	RORB
	~~BLE~~	~~CMPB~~	~~LDY~~	ROR
	~~LBLE~~	~~CMPD~~	~~LEAS~~	**RTI**
	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~CC~~	~~LBLS~~	~~CMPY~~	LSLA	SEX
A	~~BLT~~	~~COMA~~	LSLB	~~STA~~
B	~~LBLT~~	~~COMB~~	LSL	~~STB~~
	~~BMI~~	~~COM~~	LSRA	~~STD~~
A	~~LBMI~~	CWAI	LSRB	~~STS~~
IB	~~BNE~~	DAA	LSR	~~STU~~
	~~LBNE~~	~~DECA~~	MUL	~~STX~~
	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
	~~BRN~~	~~EXG~~	~~ORA~~	SWI
	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
	~~BSR~~	~~INCB~~	~~ORCC~~	SWI3
E	~~LBSR~~	~~INC~~	**PSHS**	SYNC
	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
T	~~LBVC~~	~~JSR~~	**PULS**	~~TSTA~~
	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
	~~LBVS~~	~~LDB~~	ROLA	~~TST~~
	~~CLRA~~	~~LDD~~	ROLB	

ADDRESSING MODES

~~HERENT~~
~~RECT~~
~~TENDED~~
~~MEDIATE~~
~~MPLE INDEXED~~
~~LATIVE~~
~~SPLACEMENT INDEXED~~
~~TO INCREMENT/DECREMENT~~
DIRECT
)PHISTICATED

PSEUDO OPS

)U	ORG
~~B~~	RMB
~~C~~	SET
~~B~~	~~SETDP~~

ld Type = Present Chapter
gular Type = Future Chapters
~~lic Type~~ = *Past Chapters*

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	W(ORD) MODE
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	; FORCE NUMERIC
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	**: FORCE FLAGS**
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	**STACK OPERATIONS**
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	MULTIPLES
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Chapter 15
Using the Stack to Hold Temporary Results

The S stack is not only used to hold the return address for subroutines, it can also be used as temporary storage to supplement the cpu registers. In this chapter we'll take a look at the 6809 instructions that "push" and "pull" data from the S stack area.

Stack Uses

In the last chapter we said that the stack was used for storage of addresses when a BSR was executed to a subroutine. There are actually three uses for the stack:

- Saving return address for BSRs
- Saving temporary data for PSHS and PULS use
- Saving the interrupt point for interrupts

We've already described what happens in BSRs, but we'll talk about the other two uses here.

Interrupts

One of the two uses of the stack is somewhat esoteric. Interrupts are external or internal inputs to the system that signify a "real-world" event. One example might be an interrupt from a remote keyboard. During the time that no key is pressed, the program would run normally. When a key was pressed, however, an "external" interrupt could be generated that would cause the 6809 cpu to stop processing after the current instruction, and to jump to a special interrupt processing routine.

The interrupt processing routine is simply another assembly-language program that would take the required action for the external interrupt. In this case, the remote keyboard interrupt handler would probably read in the character from the keyboard and store it in a "buffer," or storage area.

The interrupt action itself causes the interrupt point to be put into the stack in a very similar fashion to the BSR. An RTI, or Return From Interrupt, instruction at the end of the interrupt processing program would retrieve the interrupt point address from the stack and put it into the PC register in an almost identical action to a normal RTS.

Interrupts are used primarily to let the 6809 number crunch away on a "background" job, such as running a business package, while a high-priority "foreground" task infrequently interrupts for a response to its action.

Another example of interrupts is the "real-time" clock interrupt. This interrupt occurs every 16 2/3 milliseconds or so (16.667/1000ths of a second) and causes the RTC interrupt processing routine to be entered. The RTC interrupt processing increments a count which is used to keep time in the system.

We won't be discussing interrupts any further in this book. Suffice it to say that they are used infrequently in applications programs and are most used in special systems software.

Hints and Kinks 15-1
More on Interrupts

Since you're **really** interested in interrupts we'll whet your appetite... Actually an interrupt causes more than just the the address of the next instruction to be pushed into the S stack — it also causes the Condition Codes to be saved, and for certain interrupts in the 6809 causes **all** of the cpu registers to be saved. The former is a "fast interrupt," or FIRQ, and the latter an IRQ, compatible with the 6800. The FIRQ improves the interrupt response time of the 6809.

The RTI instruction is used to return from an interrupt the same way that an RTS returns from a subroutine. The RTI pops the return address, the Condition Codes, and the other cpu registers (if an IRQ occurred) from the stack to return to the interrupted point. Where is the interrupted point? It could be anywhere. Usually interrupts occur and the "background" program is not even aware that they occurred as the "interrupt processor" saves all of the "environment" (i.e. Condition Codes and register contents).

There are two interrupt bits in the Condition Codes which enable or disable the FIRQ or IRQ interrupts. They are in bit 6 and 4 of the Condition Code.

You will never have to worry about interrupts in your programming until you get to some intermediate or advanced assembly-language programming.

PSHSes and PULSes

The remaining use for the stack, however, is used all the time. A PSHS "pushes" one or more 6809 registers onto the stack, while a PULS "pulls" one or more bytes of data from the stack and puts it into one or more registers.

Why the "PSHS" and "PULS"? The stack can be thought of as a "push-down stack" similar to a dinner plate stacker in Joe's Greasy Spoon. A PSHS or BSR pushes one or more plates onto the stack. Further plates can be pushed on top of the previously pushed plates.

A PULS or RTS "pops up" the last one or more plates pushed. Successive PULSes or RTSes pop up the stacker until no plates are left.

The "minimum" PSHS is

PSHS CC or

PSHS A or

PSHS B or

PSHS DP or

PSHS X or

PSHS Y or

PSHS U or

PSHS PC

One of the above PSHSes pushes one byte (CC, A, B, or DP) or two bytes (X, Y, U, or PC) onto the stack. What about PSHing multiple registers onto the stack? Easy, just combine all the registers you want to PSHS into one instruction; if you want to PSH CC, A, and X, do a

```
        PSHS      CC,A,X
```

with the registers specified in any order (you could do a PSHS X,A,CC, for example).

The same thing holds true for PULSes. To pull data from the stack and put it into registers, do a PULS with one or more registers specified. One or two bytes will be pulled for each register, depending upon the size of the register:

```
        PULS      A           GET A
        PULS      X,U         GET X,U
        PULS      PC,U,CC,A   GET PC,U,CC, AND A
```

Again, the registers may be specified in any order.

To keep things straight as to how the registers appear on the stack for multiple PSHSes and PULSes, there is a predefined order for data PSHSed or PULSed from the stack. The PSHS order is

```
CC,A,B,DP,X,Y,U,PC
first           last
```

The PULS order is the reverse

```
PC,U,Y,X,DP,B,A,CC
```

All this means to you as the programmer, is that you know the order in which multiple registers are pushed onto the stack, and you can be certain that any PULS is done in compatible fashion.

Hints and Kinks 15-2
Multiple PSHSes and PULSes

If you do a multiple PSHS, just be certain you know what registers are on the stack. The order is predefined by the PSHS action. When you do the corresponding PULS, you would normally use the same registers, otherwise you may pull more or less bytes than you pushed. If you don't use the same registers, the same number of bytes must be PULSed as pushed. This is all right:

```
PSHS  A,B,X
.
.
PULS  X,Y     (A,B to X, X to Y)
```

but this is not:

```
PSHS  X,Y
.
.
PULS  A,Y     (retrieves one less byte)
```

Often there is a corresponding PULS for every PSHS.

Let's see how the PSHSes and PULSes work.

```
                        00100 ***********************************************
                        00110 * SCAN TABLE FOR SMALLEST ENTRY SUBROUTINE    *
                        00120 *    ENTRY: (X)=>TABLE                        *
                        00130 *           (B)=SIZE OF TABLE 1-255,0=256     *
                        00140 *    EXIT:  (X)=>SMALLEST ENTRY IN TABLE      *
                        00150 *           (A)=SMALLEST ENTRY                *
                        00160 ***********************************************
0B5C 86    FF           00170 SCANTY  LDA     #$FF     SET UP LOWEST
0B5E B7    0B7E         00180         STA     LOWEST   SET LOWEST
0B61 A6    84           00190         LDA     ,X       GET FIRST BYTE
0B63 34    10           00200         PSHS    X        INITIALIZE STACK
0B65 A6    84           00210 SCN010  LDA     ,X         GET BYTE
0B67 B1    0B7E         00220         CMPA    LOWEST     COMPARE WITH LOWEST
0B6A 24    07           00230         BHS     SCN020     GO IF C>=A
0B6C B7    0B7E         00240         STA     LOWEST     NEW LOWEST
0B6F 35    40           00250         PULS    U          PREVIOUS PNTR
0B71 34    10           00260         PSHS    X          SAVE THIS LOC'N
0B73 30    01           00270 SCN020  LEAX    1,X        POINT TO NEXT BYTE
0B75 5A                 00280         DECB               DECREMENT COUNT
0B76 26    ED           00290         BNE     SCN010     GO IF NOT END
0B78 35    10           00300         PULS    X        GET POINTER
0B7A B6    0B7E         00310         LDA     LOWEST   LOWEST IN A
0B7D 39                 00320         RTS              RETURN
0B7E       00           00330 LOWEST  FCB     0
           0000         00340         END
00000 TOTAL ERRORS

LOWEST  0B7E
SCANTY  0B5C
SCN010  0B65
SCN020  0B73
```

Figure 15-1. Scan Table for Smallest Entry Program

If you're trying to enter this code, don't try to execute this program at this point.

This program is a complete subroutine, as we talked about in the last chapter. It is a complete set of code to perform one specific function, to scan a table for the smallest byte in the table. If we had a table consisting of the one byte entries of 45,3,47,89,100,2,3,4,56, the subroutine would find the 2 entry in the table.

It's common to divide a large programming job into many different subroutines, each performing a simple function. Another trick that's used is to make each subroutine as "generic" in nature as possible. In this case, we didn't limit ourselves to **any** table, or **any size** table. We left the location and size of the table variable!

The location and size of the table are called **parameters** that are **input** to the subroutine. On entry into the subroutine, the X index register points to the table start, and the B register holds the size of the table.

On **exit**, the X register points to the location of the smallest entry in the table and the A register holds the actual entry itself. These are the **output parameters.**

"Parameters" might be called arguments, or "gozintas" and "gozoutas."

Hints and Kinks 15-3 Subroutine Parameters

One way of passing parameters is to put the arguments in cpu registers and call the subroutine. Another way is to pass the parameters in a predefined "common" area. Sometimes the Z Condition Code is set or reset on return from the subroutine to denote a "valid" or "invalid" condition from the subroutine. The BSR and RTS instructions do not affect the Condition Codes.

Because we've made SCANTY general, we can use it to scan any table of any size and it becomes a "general-purpose" subroutine.

SCANTY uses PSHSes and PULSes for temporary storage. Let's see how it does this.

First of all, X contains a pointer to the table on entry, and B contains the size of the table. The table is made up of from 1 to 256 one-byte entries. If B is initially 0, it denotes a 256-byte table.

The first thing that SCANTY does is to pick up the first table byte. This may turn out to be the lowest-valued byte, but probably isn't. The location of this byte (location in X) is then **PSHSed** onto the stack. The stack now looks like Figure 15-2.

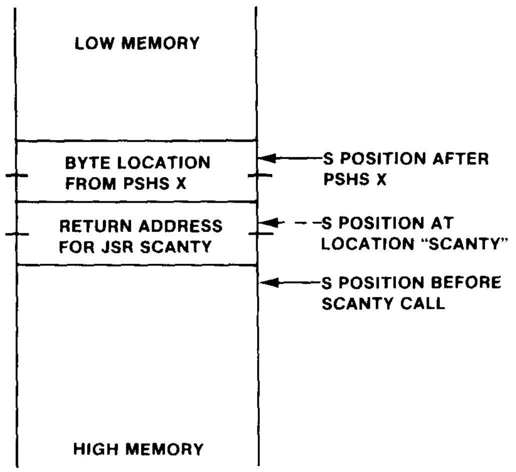

LOW MEMORY
BYTE LOCATION FROM PSHS X
RETURN ADDRESS FOR JSR SCANTY
S POSITION AFTER PSHS X
S POSITION AT LOCATION "SCANTY"
S POSITION BEFORE SCANTY CALL
HIGH MEMORY

Figure 15-2. Stack after SCANTY

The loop from SCN010 is the main (the only) loop of SCANTY. It will go through the table and compare each byte with the current low byte. If any byte is lower than the current low byte, it will be stored in variable LOWEST and its location will be saved in the stack. We're starting off with the first byte and its location as an arbritrary initial value.

In the loop: The next byte is picked up and put into A. A is then compared with LOWEST, which holds the current lowest byte.

If the next byte is greater than or equal to C, SCN020 bumps the X pointer to the next location and the count in B is decremented, causing a loop back to SCN010.

If the next byte is less than the current, it replaces the contents of LOWEST. Then, the PULS Y pops the stack and puts the current location into Y. This is done only to **reset** the stack, to get rid of the current location. Next, the PSHS X pushes the current location into the stack.

After the loop is over (the count in B goes to 0), the stack holds the location of that lowest value.

Note that when we say, "the stack holds. . .," what we really are saying is that "the stack currently holds a 16-bit data value; no corresponding PULS has been issued for it." The stack might hold three or four different 16-bit values at this point, depending upon the program. Just remember that when the stack is used for temporary storage in this way, we'll have to eventually use the values, or at least reset the stack pointer by dummy PULSes or other instructions, which we'll talk about in a later chapter.

Another interesting point: Although we used a PSHS X to save the location in the stack, there's no reason at all that we have to PULS that value back into the X register. In fact we're PULSing it into another register, the Y register. Once 8-bit values are in the stack they can be PULSed by A, B, or DP; once 16-bit values are in the stack, they can be PULSed by X, Y, U, or PC at will.

After the loop, we PULS the location into the X register, as shown in Figure 15-2D. The lowest value in LOWEST is then put into A.

The last instruction is an RTS that PULSes the return address of the calling program. The "calling program" term means that another program has called the SCANTY subroutine.

Want to see how SCANTY works? The following program shows how SCANTY might be "called" as a subroutine:

```
START  LDX    #$400     SCREEN LOCATION
       LDB    #0        1/2 OF SCREEN SIZE
       BSR    SCANTY    CALL SCANTY
LOOP   JMP    LOOP      LOOP HERE
```

You can assemble this program by inserting it before SCANTY, breakpointing location LOOP, and executing from location START.

At the end of the execution the X register should point to the lowest value in the first half of the screen area, and the A register should contain the lowest value itself. (Use the ZBUG R command to look at the registers and first half of the screen contents; however, be aware that the screen is changing as you use ZBUG! It's best to note the position of the lowest value before running SCANTY and ZBUG.)

Multiple Subroutines

We now have a standard general-purpose subroutine for searching any table of 256 bytes or less for the lowest value. What can we do with it?

One idea that comes to mind is to use it for a two-buffer sort. Remember in a previous lesson when we implemented a bubble sort? We mentioned the two-buffer sort as a sort option. The two-buffer sort requires twice the storage space because we need an extra buffer. Memory is cheap, however, so let's implement this version of a sort.

To do this, we'll need two more subroutines:

```
*STORE ENTRY IN A INTO (Y) LOCATION
STORE  STA   ,Y+        STORE ENTRY
       RTS              RETURN
*MARK OLD ENTRY WITH -1
MARK   LDA   #$FF       -1
       STA   ,X         STORE -1
       RTS              RETURN
```

The first of these, STORE, stores the contents of A into the location pointed to by Y.

The second, MARK, marks the location pointed to by X with a -1 (i.e. stores a -1 value to the location).

Given these three subroutines, we can construct a short program to sort the 256 bytes of data in the first half of the screen into the second half of the screen:

```
                       * TWO-BUFFER SORT
0DAD 108E 0500         SORT     LDY     #$400+256  SECOND 1/2 SCREEN
0DB1 C6   00                    LDB     #0         256 BYTE COUNT
0DB3 34   04           SOR010   PSHS    B          SAVE COUNT
0DB5 8E   0400                  LDX     #$400      POINT TO FIRST HALF
0DB8 C6   00                    LDB     #0         256 BYTES
0DBA 8D   0C                    BSR     SCANTY     FIND LOWEST ENTRY
0DBC 8D   2D                    BSR     STORE      STORE IN SECOND HALF
0DBE 8D   2E                    BSR     MARK       DELETE FIRST BUF ENTRY
0DC0 35   04                    PULS    B          GET COUNT
0DC2 5A                         DECB               DECREMENT
0DC3 26   EE                    BNE     SOR010     LOOP IF NOT DONE
0DC5 7E   0DC5         LOOP     JMP     LOOP
                       ***********************************************
                       * SCAN TABLE FOR SMALLEST ENTRY SUBROUTINE    *
                       *     ENTRY: (X)=>TABLE                       *
                       *            (B)=SIZE OF TABLE 1-255,0=256    *
                       *     EXIT:  (X)=>SMALLEST ENTRY IN TABLE     *
                       *            (A)=SMALLEST ENTRY               *
                       ***********************************************
0DC8 B6   FF           SCANTY   LDA     #$FF     SET UP LOWEST
0DCA B7   0DEA                  STA     LOWEST   SET LOWEST
0DCD A6   84                    LDA     ,X       GET FIRST BYTE
0DCF 34   10                    PSHS    X        INITIALIZE STACK
0DD1 A6   84           SCN010   LDA     ,X         GET BYTE
0DD3 B1   0DEA                  CMPA    LOWEST     COMPARE WITH LOWEST
0DD6 24   07                    BHS     SCN020     GO IF C>=A
0DD8 B7   0DEA                  STA     LOWEST     NEW LOWEST
0DDB 35   40                    PULS    U          PREVIOUS PNTR
0DDD 34   10                    PSHS    X          SAVE THIS LOC'N
0DDF 30   01           SCN020   LEAX    1,X        POINT TO NEXT BYTE
0DE1 5A                         DECB               DECREMENT COUNT
0DE2 26   ED                    BNE     SCN010     GO IF NOT END
0DE4 35   10                    PULS    X        GET POINTER
0DE6 B6   0DEA                  LDA     LOWEST   LOWEST IN A
0DE9 39                         RTS              RETURN
0DEA      00           LOWEST   FCB     0        HOLDS LOWEST AT END
                       * STORE ENTRY IN A INTO (Y) LOCATION
0DEB A7   A0           STORE    STA     ,Y+      STORE ENTRY
0DED 39                         RTS              RETURN
                       * MARK OLD ENTRY WITH -1
0DEE 86   FF           MARK     LDA     #$FF     -1
0DF0 A7   84                    STA     ,X       STORE -1
0DF2 39                         RTS              RETURN
          0000                  END
00000 TOTAL ERRORS

LOOP     0DC5
LOWEST   0DEA
MARK     0DEE
SCANTY   0DC8
SCN010   0DD1
SCN020   0DDF
SOR010   0DB3
SORT     0DAD
STORE    0DEB
```

Figure 15-3. Two Buffer Sort

This sequence uses the three subroutines to do the sort. The first, SCANTY, finds the location and value of the lowest entry in the first screen half buffer.

The second, STORE, stores the value in A into the second screen half buffer by using Y as a pointer.

The third, MARK, sets the original location of the current value in the first screen half buffer to -1. A value of -1 is used as a flag to say "this location already was a lower value and is not be used from this point."

Before the BSR to SCANTY, X and B must be initialized to point to the first screen half buffer and for a count of 256, respectively.

About the only "tricky" part of SORT is using a PSHS B to save the count before the BSR to SCAN is made. If this were not done, the loop count in B would be destroyed by the table size parameter. After the three subroutines, the count in B is restored by a PULS B. This type of operation is very common in saving registers.

If you run SORT, you will see the second half of the screen fill up with ordered characters, and the first half of the screen fill up with -1 characters, which display as an orange rectangle in the character position. By the way, this SORT also works with 0FFH values (-1). Do you see why?

Hints and Kinks 15-4
Screen Graphics

The -1 character output to the text screen defaults to "semigraphic 4" mode, where there are 4 "elements" per character position and from 1 to 8 colors. The format is 1CCCEEEE, where 1 marks the byte as graphics, CCC is a color code from 000 through 111 and EEEE is the on/off status of the four elements. A -1 (11111111) sets orange (111) and turns on all elements (1111). See "Color Computer Graphics" for more on this.

We'll be using PSHSes and PULSes frequently from this point on. In fact, we've just scratched the surface of the use of stacks in the 6809 and Color Computer!

Hints and Kinks 15-5
To Examine the Condition Codes

The colon key lets you convert to "Condition Code Format." If you have an 8-bit value that represents Condition Codes (as in the stack from a PSHS CC) and you are examining the location containing the CC value, do a colon after the display to convert to the CC mnemonics:

```
#3000/ 05 :=ZC
```

Review

To review what we've learned here:

- Interrupts use the stack by pushing the return address for the interrupt point in the stack
- PSHSes and PULSes are used to temporarily store data in the stack
- Any combination of registers — CC, A, B, X, Y, U, or S can be PSHSed or PULSed
- There must be a PULS executed for every PSHS executed, or the stack must be reset by "dummy" PULSes or other instructions that adjust the S register
- Subroutines are often small segments of "generalized" code that work for many different sets of conditions
- Parameters are often passed to and from subroutines; these parameters are "arguments" that define the conditions for the subroutine
- Once data is in the stack, it is not related to any register; the same register or a different register may be used to retrieve the data or the data may be simply discarded

For Further Study

RTI, RTS action (Appendix I)
PSHS, PULS formats (Appendix II)

KEY CHART — CHAPTER 16

INSTRUCTIONS

~~LBHS~~	~~CLRB~~	~~LDS~~	**ROL**
~~BITA~~	~~CLR~~	~~LDU~~	**RORA**
~~BITB~~	~~CMPA~~	~~LDX~~	**RORB**
~~BLE~~	~~CMPB~~	~~LDY~~	**ROR**
~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~LBLS~~	~~CMPY~~	**LSLA**	**SEX**
~~BLT~~	~~COMA~~	**LSLB**	~~STA~~
~~LBLT~~	~~COMB~~	**LSL**	~~STB~~
~~BMI~~	~~COM~~	**LSRA**	~~STD~~
~~LBMI~~	CWAI	**LSRB**	~~STS~~
~~BNE~~	DAA	**LSR**	~~STU~~
~~LBNE~~	~~DECA~~	**MUL**	~~STX~~
~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~BSR~~	~~INCB~~	~~ORCC~~	SWI3
~~LBSR~~	~~INC~~	~~PSHS~~	SYNC
~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~LBVS~~	~~LDB~~	**ROLA**	~~TST~~
~~CLRA~~	~~LDD~~	**ROLB**	

ADDRESSING MODES

~~ERENT~~
~~ECT~~
~~ENDED~~
~~EDIATE~~
~~PLE INDEXED~~
~~ATIVE~~
~~LACEMENT INDEXED~~
~~O INCREMENT/DECREMENT~~
RECT
HISTICATED

PSEUDO OPS

ORG
RMB
SET
~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
C(ONTINUE)	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	**W(ORD) MODE**
~~G(O)~~	~~X BREAKPOINT~~
H(ALF) SYMBOLIC	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
M(NEMONIC) MODE	→ BRANCH INDIRECT
N(UMERIC) MODE	**; FORCE NUMERIC**
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
S(YMBOLIC DISPLAY)	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	**ROTATES, SHIFTS**
~~ADDITION AND SUBTRACTION~~	**MULTIPLES**
~~CONDITION CODES~~	DIVIDES
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Type = **Present Chapter**
lar Type = Future Chapters
~~Type~~ = *Past Chapters*

Chapter 16
Rotates, Shifts, and Multiplication

In this chapter we'll be looking at a number of related instructions that move data by "shifting." In some shift instructions, data is "rotated" back into the memory byte or cpu register; in other shifts, zeroes fill the vacated positions. Simple multiplies can be done by shifting. We'll also look at the powerful MUL instruction which is a built-in "hardware" multiply.

Rotates

There are two rotate instructions, ROL and ROR. Like the logical instructions, the rotates work on either the A or B accumulators or a byte in a memory location.

The instructions rotate the register or memory location either to the right or left one bit at a time, as shown in Figure 16-1. However, the C Condition Code is also rotated along with the Carry, making the rotate a 9-bit rotate.

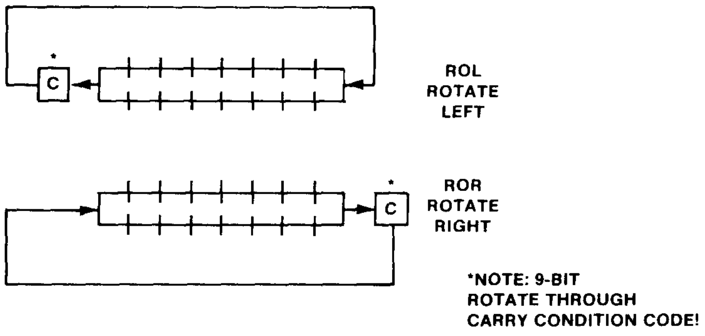


Figure 16-1. Rotate Instructions

To see how these work look at the following program:

```
                   00100 * ROL, ROR OPERATION
0953 86     A5     00110 ROTDEM  LDA     #$A5      10100101
0955 C6     09     00120         LDB     #9        9 TIMES
0957 49            00130 ROT020  ROLA                ROTATE
0958 5A            00140         DECB                DECREMENT COUNT
0959 26     FC     00150         BNE     ROT020      GO IF NOT END
095B 86     A5     00160         LDA     #$A5      10100101
095D C6     09     00170         LDB     #9        9 TIMES
095F 46            00180 ROT030  RORA                ROTATE
0960 5A            00190         DECB                DECREMENT COUNT
0961 26     FC     00200         BNE     ROT030      GO IF NOT END
0963 7E     0963   00210 LOOP    JMP     LOOP      LOOP HERE
            0000   00220         END
00000 TOTAL ERRORS

LOOP     0963
ROT020   0957
ROT030   095F
ROTDEM   0953
```

Figure 16-2. ROL, ROR Operation Program

Assemble the program. Now go to ZBUG and breakpoint at location ROT020, ROT030, and LOOP. Execute from ROTDEM. When you reach the breakpoint, do an R and look at the contents of A. Now do a ZBUG C to continue from the breakpointed location. You'll hit the breakpoint again, and you can look at the contents of A again. Continue until you hit the breakpoint at ROT030. Continue (C) from that point. What you'll see is the rotate action of ROLA and RORA. It'll look like this:

	A Contents(Hex)	A Contents (Binary)	Carry
ROT020:	A5	10100101	X
		0100101X	1
		100101X1	0
		00101X10	1
		0101X101	0
		101X1010	0
		01X10100	1
		1X101001	0
		X1010010	1
	A5	10100101	X
ROT030:	A5	10100101	X
		X1010010	1
		1X101001	0
		01X10100	1
		101X1010	0
		0101X101	0
		00101X10	1
		100101X1	0
		0100101X	1
	A5	10100101	X

In the first part, the A register is rotated left, one bit at a time. As each bit is shifted around from bit 7 back into A through bit 0, you should also see the bit going into the Carry Condition Code. Nine shifts are done and at the end of the time, the A register should have the original value of $A5, 10100101.

The ROT030 loop will rotate the A register to the right, one bit at a time. As each bit is shifted around from bit 0 back into A through bit 7, you'll also see the bit going into the Carry Condition Code. Nine shifts are done. At the end of the shifts, A will have the original value.

The N and Z Condition Codes are also affected for these two rotates. The Z Condition Code is set if the shifting results in a zero. (If the number is non-zero, Z will never be set.) The N Condition Code is set to the sign of the shifted result.

Hints and Kinks 16-1
Rotates

Rotates are useful for testing bits in a cpu register or memory byte through 9 iterations without destroying the original value.

There are several other types of shifts in the 6809 that are used frequently. One of these is the "logical" shift, and the second is the "arithmetic" shift.

Logical Shifts

Logical shifts are different from rotates because they do not recirculate the data in the register or memory location. Logical shifts shift in zeroes in place of the data from the other end of the register or memory location as shown in Figure 16-3.

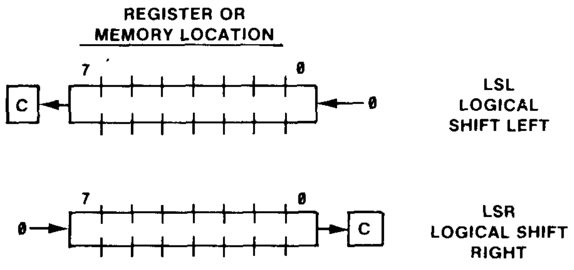


Figure 16-3. Logical Shifting

The two logical shifts in the 6809 are the LSR (Logical Shift Right) and LSL (Logical Shift Left).

As in the case of the rotates, any bit leaving the register or memory location as a result of the shift goes into the Carry Condition Code. All Condition Codes with the exception of the "Half Carry" Condition Code are set on the result of the shift. (The "Half Carry," H, is the Carry out of bit 4 and is used for certain "binary-coded-decimal" operations.)

If we had 01110001 in the B register, then a

```
        LSRB            SHIFT B RIGHT
```

would result in a 00111000 with the Carry Condition Code set to 1. Seven LSRBs would result in 00000000, clearing the B register.

The LSL instruction works exactly the same way in reverse. If we had 01110001 in B, then

```
        LSLB            SHIFT B LEFT
```

would result in 11100010, with the Carry Condition Code set to 0.

LSR and LSL can be used with the A and B registers and with any byte in memory.

```
            LSL     $3000
```

for example, shifts the contents of RAM location $3000 left one bit logically.

Multiplying and Dividing By Shifting

Every time a logical shift left is done, the original value in the register or memory is multiplied by 2:

```
        00110010    Original=50
  <=    01100100    After LSL=100
  <=    11001000    After LSL=200 (absolute)
  <=    10010000    After LSL=144 (invalid)
```

Every time a logical shift right is done, the original value is divided by 2:

```
        00110010    Original=50
  =>    00011001    After LSR=25
  =>    00001100    After LSR=12
  =>    00000110    After LSR=6
  =>    00000011    After LSR=3
```

You can see from the above examples that there is a limit to the number of multiplies by shifting that can be done. This limit is the size of the register or memory location itself. After a certain point, the results are invalid.

Another interesting point is that a divide by shifting to the right results in a loss of the remainder. Actually, a divide by shifting puts the remainder (0 or 1) into the Carry Condition Code, but it is lost on the next shift.

Multiplying and dividing by shifting, then, can be done for small values and with an eye on the limitations of this type of processing.

Multiplication by powers of two can also be done by adding either 8-bit values or 16-bit values to themselves.

Hardware Multiplies

What about multiplication or division by other than powers of two? Suppose we wanted to multiply any 8-bit number by any other 8-bit number? There is a built-in "hardware" multiply in the 6809. However, there is no built-in divide! Lack of a divide is not an unusual situation for microcomputers, by the way. Most other microprocessors, such as the 8080, Z-80, 6502, and 6800 (predecessor of the 6809) not only don't have a hardware divide, they also do not have a hardware multiply!

The Multiply instruction in the 6809 is called "MUL" and multiplies the contents of the A register by the contents of the B register. The result goes into the D register (A contains the "most significant" 8 bits, while B contains the "least significant" 8 bits). The process is shown in Figure 16-4.

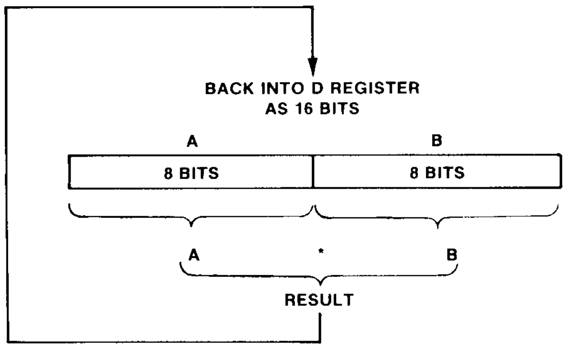


Figure 16-4. MUL Action

Suppose we want to use the MUL to multiply the following numbers:

1)	\$50 times \$02	(80 * 2)
2)	\$0A times \$14	(10 * 20)
3)	\$00 times \$00	(0 * 0)
4)	\$7F times \$02	(127 * 2)
5)	\$80 times \$02	(128 * 2)
6)	\$FF times \$FF	(255 * 255)

You can easily do this by assembling the short program

```
START   MUL
LOOP    JMP      LOOP
```

breakpointing at LOOP, and using the R command to examine the contents of the D register. Use the A/ and B/ commands in ZBUG to load the registers before the Multiply.

Did you try them? Let's do some analysis about the multiply. We're multiplying an 8-bit number by an 8-bit number. How large will the result be? That's fairly easy to figure out.

We know that in 8 bits we can hold 0 through 255 if the numbers are "unsigned" or absolute. The result of the multiply can therefore be anything from 0 (0 * 0) through 65,025 (255 * 255).

Since a 16-bit number can hold 0 through 65,535, it looks as if we should have no trouble fitting the quotient in the 16-bit HL register pair. Here is what you should have seen after the multiplies:

1) $50 times $02=$00A0 (80 * 2)=160

2) $0A times $14=$00C8 (10 * 20)=200

3) $00 times $00=$0000 (0 * 0)=0

4) $7F times $02=$00FE (127 * 2)=254

5) $80 times $02=$0100 (128 * 2)=256

6) $FF times $FF=$FE01 (255 * 255)=65,025

The MUL instruction is an "unsigned" multiply. Multiplying $FF by $FF in a "signed" or two's complement multiply would have resulted in +1, or 0000000000000001 ($01), but here the product was $FE01, or 65,025.

What about multiplies of more than 8 bits by 8 bits? After all, there's not a great deal of "significance" in an 8 bit value.

Hints and Kinks 16-2
How Fast is MUL?

The MUL executes in about 12.5 microseconds (millionths of a second), about 10 times faster than the equivalent "software" multiply!

A 16 by 8 Multiply

Here's a method for a 16 by 8 multiply. Let's say that one operand is in location OPA and OPB (2 bytes) and the other operand is in location OPC. We want to put the product in RESMSB, RESNSB, and RESLSB (3 bytes). We'll use the MUL instruction to do the multiply. How can this be done with an 8 by 8 multiply?

One way to do it is to use the fact that any 16-bit number of the form $XXYY is really XX*256+YY. If we call the two bytes of the first number AB and the byte of the second number C, then we have (A*256+B)*C. This is equal to A*C*256 + C*B, which is the same as

A*C*256=A*C shifted left 8 bits
+C*B

All we have to do is 2 separate multiplies of A*C and C*B, do some shifting and addition and we'll have the result:

```
                        00100 * 16 BY 8 MULTIPLY BY PARTIAL PRODUCTS
0AB2 B6    0AE6         00110 MUL16     LDA     OPC      GET C
0AB5 7F    0AE1         00111           CLR     RESMSB   CLEAR RESULT AREA
0AB8 7F    0AE2         00112           CLR     RESNSB   CLEAR SOME MORE
0ABB 7F    0AE3         00113           CLR     RESLSB   CLEAR THE LAST
0ABE F6    0AE4         00120           LDB     OPA      GET A
0AC1 3D                 00130           MUL              A*C
0AC2 FD    0AE1         00140           STD     RESMSB   SAVE
0AC5 B6    0AE5         00150           LDA     OPB      GET B
0AC8 F6    0AE6         00160           LDB     OPC      GET C
0ACB 3D                 00170           MUL              C*B
0ACC F3    0AE2         00180           ADDD    RESNSB   LS BYTE
0ACF 34    01           00190           PSHS    CC       SAVE CARRY
0AD1 FD    0AE2         00200           STD     RESNSB   C*B
0AD4 B6    0AE1         00210           LDA     RESMSB   GET MS BYTE
0AD7 35    01           00220           PULS    CC       GET CC
0AD9 B9    00           00230           ADCA    #0       ADD IN CARRY
0ADB B7    0AE1         00240           STA     RESMSB   STORE RESULT
0ADE 7E    0ADE         00241 LOOP      JMP     LOOP     LOOP HERE AT END
0AE1       00           00250 RESMSB    FCB     0        RESULT MS BYTE
0AE2       00           00260 RESNSB    FCB     0        RESULT NEXT SIG BYTE
0AE3       00           00270 RESLSB    FCB     0        RESULT LS BYTE
0AE4       00           00280 OPA       FCB     0        MS OF MULTIPLICAND
0AE5       00           00290 OPB       FCB     0        LS OF MULTIPLICAND
0AE6       00           00300 OPC       FCB     0        MULTIPLIER
           0000         00310           END
00000 TOTAL ERRORS

LOOP     0ADE
MUL16    0AB2
OPA      0AE4
OPB      0AE5
OPC      0AE6
RESLSB   0AE3
RESMSB   0AE1
RESNSB   0AE2
```

Figure 16-5. Sixteen by Eight Multiply Program

This code will generate a result as a 3-byte number in RESMSB through RESLSB. The most significant byte of the result will be in location RESMSB, the next in RESNSB, and the last in RESLSB. The 16-bit multiplicand must be in OPA, OPB and the 8-bit multiplier in OPC.

This technique can be used for 16 by 16 multiplies or even greater, but does get fairly tedious for a larger number of bytes. **Floating-point** representation is generally used for large numbers, as in BASIC single-precision and double-precision variables.

Arithmetic Shifts

We've gotten off on a tangent here discussing multiplies, but we really couldn't do them justice before discussing shifts.

We mentioned another type of shift at the beginning of this lesson called the "arithmetic shift." To see how this shift works, enter the following code and assemble:

```
*ARITHMETIC SHIFTS
ASHFT  LDA    #$85    LOAD A WITH 10000101
       LDB    #8      LOOP COUNT
ASH010 ASRA             SHIFT A RIGHT, ARITH
       DECB             DECREMENT COUNT
       BNE    ASH010    GO IF NOT 8
LOOP   JMP    LOOP    LOOP HERE
       END            END
```

If you assemble and execute using ZBUG to examine the B register, you should have seen the B register change as follows:

```
10000101

11000010

11100001

11110000

11111000

11111100

11111110

11111111

11111111
```

Now change the 85H to 75H and execute. You should see:

```
01110101

00111010

00011101

00001110

00000111

00000011

00000001

00000000
```

It appears that in the first case the shift "extends" ones, while in the next case zeroes are "extended." Why?

The arithmetic shift is used for signed values. When the most significant bit is a sign (or even if it isn't), the ASR will extend the msb to the right as the number is shifted. For positive numbers (sign bit=0), this works the same as an LSR, extending zeroes. For negative numbers, though, the result is different.

Looking back on the shift of 85H, let's take the two's complement of the result and see what we get:

10000101	-123
11000010	-62
11100001	-31
11110000	-16
11111000	-8
11111100	-4
11111110	-2
11111111	-1

Aha! Looks like the ASR can be used to "sign extend" the result. "Sign extend" means that the result will be shifted right with the sign intact. If we used just an LSR shift, we'd have an invalid result as in

10000101	-123
01000010	+66

Again, as in the case of an LSR, we lose a portion of the result on the shift if the number is odd — the -123 became a -62, for example. The ASR is handy, though, for those cases where we want to shift a negative number and do it with the sign properly adjusted.

There's another arithmetic shift, the arithmetic shift left, or ASL. In fact, this is identical to an LSL, as it has the same operation code. Assemble this code to compare the "opcodes" for both shifts:

```
ASLA
ASLB
ASL     $3000
LSLA
LSLB
LSL     $3000
```

Hints and Kinks 16-3
The SEX Instruction

Any company that labels their listings "AUSTIN TEXAS — MICROCOMPUTER CAPITAL OF THE WORLD" surely has the chutzpah to call a Sign Extend instruction SEX, rather than the timid SGN, SIX, or SED. And they did!

Actually, it's a rather bland instruction; it sign extends the sign bit of the B register into A. It's handy for setting up an ADDD and other operations. If A was 00000000 and B was 10010000, for example, the result would be 11111111 in A and 10010000 in B; the same number, but in 16 bits. We thought we'd mention it here because of the related ASR function.

Hints and Kinks 16-4
More on Examination and Display Modes

There are a number of ZBUG commands we haven't covered relating to the "examination" mode and "display" mode of ZBUG.

You've used the B(yte) command to enable the examination of a single byte at a time and A(SCII) to enable examination in ASCII.

W(ord) enables the examination of two bytes at a time.

```
#W
#2800/ 0852
```

The "default" (entry) examination mode is M(nemonic) which results in examination in instruction mnemonic form. If ZBUG cannot find an instruction to match the data in memory, question marks are displayed.

```
#M
#3000/ STA >3010
```

All of the above are "examination" mode commands that are mutually exclusive; only one is in force at any time.

The display mode commands are another set of functions from the examination mode commands; they determine how the location values will be displayed.

If N(umeric) mode is set, all locations will be displayed in numeric form:

```
#N
#3000/ B7
```

If S(ymbolic) mode is set, all locations will be displayed as symbolic expressions based upon the in-memory assembly, if any.

```
#S
#START+7/ STA >TABLE
 START+10/ LDA #20
```

If H(alf Symbolic) mode is set, the memory address operand is displayed in numeric, but the examination location is in full symbolic form.

```
#H
#START+7/ STA >3010
```

These modes do not affect the input format; you can still "open" a location in symbolic form even in numeric mode:

```
#N
#START+11/ BRAAB >3010
```

(The BRAAB instruction above, by the way, stands for the seldom used "Branch and Bomb.")

The "one-time" semicolon key forces a numeric display mode as in:

```
#S
#START+7/ STA >TABLE ;STA >3000
```

Review

To recap this chapter:

- Rotates rotate either the A register or B register or a memory location.
- Rotates recirculate the bits back into the register or memory location from the other end
- The Carry Condition Code is affected by the rotate along with the Z and N Condition Codes
- The Carry Condition Code is always set to the state of the bit shifted out on a rotate
- The LSR and LSL are "logical" shifts that shift a register or memory location to the right or left one bit at a time
- Logical shifts affect the N, Z, V, and C Condition Codes
- Logical shifts to the right divide by 2 and to the left multiply by 2
- The MUL instruction performs an "8 by 8" multiply of A and B, with the 16-bit product going into the D register
- The product of binary multiplies will not exceed the total number of bits in both operands
- The ASR arithmetic shift sign extends the A or B register or memory location contents as the shift is done

For Further Study

KEY CHART — CHAPTER 17

INSTRUCTIONS

~~ABX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~ADCA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~ADCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~ADDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~ADDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~ANDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~ANDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ANDCC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~ASLA~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~ASLB~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
~~ASL~~	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~ASRA~~	~~LBMI~~	CWAI	~~LSRB~~	~~STS~~
~~ASRB~~	~~BNE~~	DAA	~~LSR~~	~~STU~~
~~ASR~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~BCC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BCS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	~~ORA~~	SWI
~~LBEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	SWI2
~~BGE~~	~~BSR~~	~~INCB~~	~~ORCC~~	SWI3
~~LBGE~~	~~LBSR~~	~~INC~~	~~PSHS~~	SYNC
~~BGT~~	~~BVC~~	~~JMP~~	PSHU	~~TFR~~
~~LBGT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	PULU	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
~~DISPLACEMENT INDEXED~~
~~AUTO INCREMENT/DECREMENT~~
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	ORG
~~FCB~~	RMB
~~FCC~~ →	SET
~~FDB~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABL~~
~~/LP LINE PRINTER~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOC~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDIN~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	→ BRANCH INDIRECT
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BY~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	**DIVIDES**
~~SYMBOLIC ADDRESSING~~	DECIMAL ARITHMETIC
~~JUMPS, BRANCHES~~	BASIC INTERFACING
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Bold Type = **Present Chapter**
Regular Type = Future Chapters
~~*Italic Type*~~ = *Past Chapters*

Chapter 17
An Unsigned Divide and Signed Multiplies and Divides

Unfortunately, the 6809 has no built-in divide as it has a multiply. This is not an oversight. It takes a good deal of hardware "logic" to implement a divide. In this chapter we'll see how an "unsigned" divide of 16 bits by 8 bits can be implemented in software. We'll also look at the question of "signed" divides and multiplies and find out how to implement them.

An Unsigned Divide

Look at the following code:

```
                      00100 ******************************************************
                      00110 * DIVIDE 16 BY 8 SUBROUTINE, UNSIGNED                *
                      00120 *     ENTRY: (X)=16-BIT DIVIDEND                     *
                      00130 *            (A)=8-BIT DIVISOR                       *
                      00140 *     EXIT:  (X)=QUOTIENT                            *
                      00150 *            (A)=REMAINDER                           *
                      00160 ******************************************************
0C83 34     12        00170 DIV168  PSHS   X,A      DIVIDEND, DIVISOR
0C85 4F               00180         CLRA            CLEAR 1/2 OF DIVIDEND
0C86 E6     61        00190         LDB    +1,S     GET MSB OF DIVIDEND
0C88 8D     0B        00200         BSR    DIVIDE   DO 8 DIVIDES
0C8A E7     61        00210         STB    +1,S     REPLACE 1ST 1/2
0C8C E6     62        00220         LDB    +2,S     GET LSB OF DIVIDEND
0C8E 8D     05        00230         BSR    DIVIDE   DO 8 DIVIDES
0C90 E7     62        00240         STB    +2,S     REPLACE 2ND 1/2
0C92 35     14        00250         PULS   B,X      DISCARD DIVISOR, GET Q
0C94 39               00260 ENDEX   RTS             RETURN
0C95 8E     0008      00270 DIVIDE  LDX    #8       SETUP COUNTER
0C98 58               00280 DIV010  LSLB              SHIFT D LEFT ONE BIT
0C99 49               00290         ROLA
0C9A CA     01        00300         ORB    #1         PRESET Q BIT TO 1
0C9C 24     04        00301         BCC    DIV015     GO IF C=0
0C9E A0     62        00302         SUBA   +2,S       SUBTRACT MUST GO
0CA0 20     08        00303         BRA    DIV020     CONTINUE
0CA2 A0     62        00310 DIV015  SUBA   +2,S       DO SUBTRACT
0CA4 24     04        00320         BHS    DIV020     GO IF + OR 0
0CA6 C4     FE        00330         ANDB   #$FE       RESET Q BIT
0CA8 AB     62        00340         ADDA   +2,S       RESTORE
0CAA 30     1F        00350 DIV020  LEAX   -1,X       DECREMENT COUNT
0CAC 26     EA        00360         BNE    DIV010     GO IF NOT 0
0CAE 39               00370         RTS             RETURN TO CALLING PROG
            0000      00380         END
00000 TOTAL ERRORS

DIV010  0C98
DIV015  0CA2
DIV020  0CAA
DIV168  0C83
DIVIDE  0C95
ENDEX   0C94
```

Figure 17-1. Divide 16 By B Program

DIV168 divides a 16-bit number by an 8-bit number. This is an **unsigned** divide just as the MUL instruction performs an unsigned multiply. Some typical values might be these:

$FFFF/$01=$FFFF	$00	65,535/1=65,535 remainder 0
$F000/$20=$0780	$00	61440/32=1920 remainder 0
$03E8/$75=$0008	$40	1000/117=8 remainder 64
$0003/$30=$0000	$03	3/48=0 remainder 3

How does the DIVIDE work? The general method used in divides is "restoring division."

A paper and pencil division is shown in Figure 17-2. The divisor (the number that goes "into" the dividend) is tried with the first digit of the dividend. If this doesn't "go," the next two digits of the dividend are tried. If this doesn't work, the next three digits of the dividend are tried.

```
          43|23915 ?


                 0000001000101100 QUOTIENT OF 556
        00101011|0101110101101011
                 -0101011
                  00000111011
                    -00101011
                     0001000001
                      -00101011
                       000101100
                       -00101011
                        0000000111 REMAINDER OF 7
```

Figure 17-2. Paper and Pencil Division

If the divisor does "go," it is subtracted from the dividend. The next digit is then brought down with the result, and the process is repeated.

What we are doing in out heads is to make the determination that the divisor will "go" into the next dividend "residue." In some cases we actually try a quotient digit and find that the result is too large to be subtracted from the residue; it would give a negative number. In these cases we "restore" the original residue and try again.

When implemented in a software divide, the program **always** subtracts the divisor from the residue, as shown in Figure 17-3. If the result of the subtraction is negative, the subtraction won't "go." In this case the residue is "restored" by adding back the divisor, and the quotient (result) bit is set to 0. If the subtraction does go, the quotient bit is set to 1, and no "restore" is done.

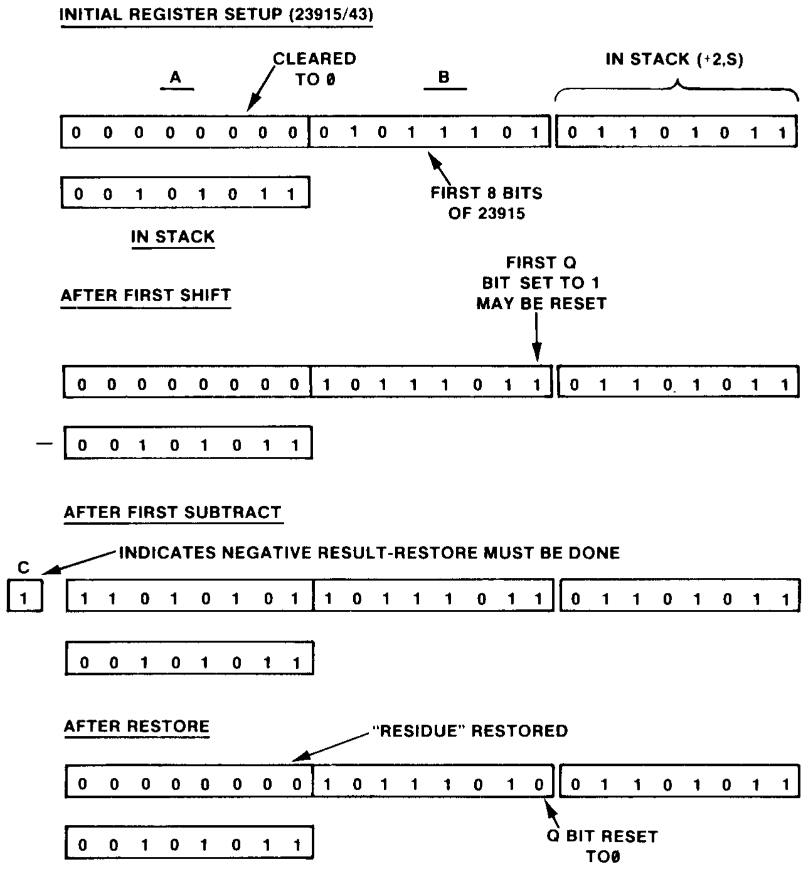


Figure 17-3. Software Divide Algorithm

The DIV168 program shown above uses the D register to hold the dividend, as shown in Figure 17-3. The divisor is subtracted from the dividend for 16 "iterations" of the divide. After the first iteration, the dividend becomes the "residue," not really the dividend any longer but a portion of it. The divisor is held in the stack.

The residue in D is shifted left 1 bit for each of the 16 iterations.

There will be a maximum of 16 bits in the quotient, as in $FFFF divided by 1. Therefore there will be 16 subtracts, each one generating a quotient bit of 0 or 1.

The quotient bit goes into the least significant bit of the D register (least significant bit of B). It is set to a 1 before the subtract, and reset if the subtract doesn't go. The quotient bit can be put into this bit because the D register leaves a vacated bit as it is shifted left.

There are two parts to DIV168, the DIVIDE subroutine, and the "main" code from DIV168 to ENDEX.

The DIVIDE subroutine does 8 subtracts. The X register is used as a loop counter, and is loaded with an initial count of 8. Next, the D register is shifted left one bit by doing a logical left shift of B (0 into the right, most significant bit out to Carry) followed by a Rotate left shift of A (Carry from B goes into least significant bit and most significant bit of A goes into the Carry.) At the same time, the Q bit in the least significant bit of B is set to 1; this 1 may be changed to a 0 later if the divide "doesn't go."

First a check is made of the Carry. If the Carry is a 1, the high order bit of the dividend is a 1, and the subtract "must go." In this case, the subtract is done to adjust the dividend and a BRA DIV020 is done.

Now the subtract of the divisor from the residue is done by SUBA +2,S. The stack pointer S at this time points to 2 more than the divisor since the BSR resulted in storing the return address. A "+2,S" picks up the divisor, which was stored in the stack by the initial PSHS X,A.

If the subtract "goes," the result in A is either 0 or a positive residue, and the BHS branches to DIV020. If the subtract doesn't go, the Q bit in B is reset to 0 by the ANDB #$FE, and a restore of the original value in A is done by adding back the divisor.

We're now at DIV020. The count in X is decremented by one. If this is not 0, there are more of the 8 iterations to perform, and a branch to DIV010 is done, otherwise a return is made back to the main code in DIV168.

The main code in DIV168 calls DIVIDE in two steps, the first to find the 8 quotient bits for the divide of the 8 **most** significant bits of the dividend, and the second to find the 8 quotient bits for the divide of the 8 **least** significant bits of the dividend. Breaking the divide up this way is necessary because we don't have enough registers to do 16 iterations at the same time!

For each call to DIVIDE, the 8 bits of the dividend are picked up from the stack, the DIVIDE is called, and the quotient bits from the DIVIDE are then stored back into the stack in place of the dividend 8 bits. At the end of the divide, the PULS discards the divisor and puts the quotient in the X register (PULS B,X). The last "residue" in A is the remainder.

If you want to enter and execute DIV168, use ZBUG to set X and A to sample values for the divide and then execute from DIV168 with a breakpoint at ENDEX. You might also try to breakpoint at DIV020 with a C(ontinue) to see how the registers change for each of the 16 iterations of DIV168.

DIVIDE is a typical divide in software. It is an "unsigned" divide that does not work with two's complement numbers.

There is one case in the DIVIDE which you should be aware of. This is division by zero. If you divide $FFFF by $01 (65,535/1), you'll get a quotient of $FFFF, which is correct. What do you get when you try dividing $FFFF by $00? Try it and see. What about $03F8 by $00?

Both of these cases and any division by 0 produces $FFFF, which is incorrect. Because of this, using a 0 divisor is not allowed in divides, and is not allowed in most other mathematics operations.

Hints and Kinks 17-1
How Fast Is the Divide?

Actually we used the wrong title here. It should have been "How Slow is the Divide?." Software multiplies and divides are not known for speed compared to their hardware counterparts.

We haven't accurately timed the DIV168, but we can estimate as follows: The DIVIDE subroutine has 8 iterations of 9 instructions. Thats 72 instructions total. It's done twice, and that's 144 instructions. There are 2 instructions in DIVIDE that are also executed twice, and that's 148 instructions. The DIV168 "main line" code has 10 instructions, and that's 158 instructions.

An average 6809 instruction in the Color Computer executes in about 4 "cycles" (see Appendix II). Each cycle is about 1.124 microseconds, so we have 4*1.124*158=710 microseconds, or about 1400 divides per second, compared to 80,800 multiplies using the hardware MUL instruction! Can DIV168 be speeded up? Yes, and we'll leave it up to you as an exercise...

Dividing by Larger Numbers

The DIV168 above is about the minimum-sized divide that is still useful. By proper "scaling" you can use DIVIDE to get fairly good accuracy. For example, if you wanted to add 1/2 + 1/4 + 1/8, and so forth, you could do the divide as 10000/2 + 10000/4 + 10000/8... The resulting quotient would be "scaled up" to 10,000 times the actual result, and you could find the true result by putting a decimal point four places in front of the computed result:

```
10000/2=      5000
10000/4=      2500
10000/8=      1250
10000/16=      625
10000/32=      312
10000/64=      156
10000/128=      78
              ----
              9921   => .9921 actual is .9921875
```

Doing "Signed" Multiplies and Divides

In this chapter and the last we've discussed **unsigned** multiplies and divides. How would you do "signed" multiplies and divides?

Although it's possible to do them directly with special multiplies and divides, the easiest way is to first convert the operands to absolute values, do the multiply or divide, and then change back the result to the proper sign. An example is shown for DIVIDE:

```
                    * SIGNED DIVIDE
0F3E B6     0F6E    SDIV    LDA     DIVDND   GET DIVIDEND
0F41 B8     0F70            EORA    DIVSOR   EOR DIVISOR
0F44 34     01              PSHS    CC       SAVE RESULT SIGN
0F46 FC     0F6E            LDD     DIVDND   TEST SIGN OF DIVIDEND
0F49 2A     06              BPL     SDI010   GO IF +
0F4B CC     0000            LDD     #0       FIND ABSOLUTE VALUE
0F4E B3     0F6E            SUBD    DIVDND   NEGATE
0F51 1F     01      SDI010  TFR     D,X      DIVIDEND NOW IN X
0F53 B6     0F70            LDA     DIVSOR   GET DIVISOR
0F56 2A     01              BPL     SDI020   GO IF +
0F58 40                     NEGA             FIND ABSOLUTE VALUE
0F59 8D     16      SDI020  BSR     DIV168   DO UNSIGNED DIVIDE
0F5B 35     01              PULS    CC       GET RESULT SIGN
0F5D 2A     0C              BPL     LOOP     GO IF ALL OK
0F5F 34     12              PSHS    A,X      QUOTIENT TO STACK
0F61 CC     0000            LDD     #0       CLEAR D
0F64 A3     61              SUBD    +1,S     NEGATE QUOTIENT
0F66 1F     01              TFR     D,X      QUOTIENT BACK TO X
0F68 35     22              PULS    A,Y      DROP X, GET A
0F6A 40                     NEGA             NEGATE REMAINDER
0F6B 7E     0F6B    LOOP    JMP     LOOP     LOOP HERE
0F6E        0000    DIVDND  FDB     0        DIVIDEND
0F70        00      DIVSOR  FCB     0        DIVISOR
                    ***************************************************
                    * DIVIDE 16 BY 8 SUBROUTINE, UNSIGNED             *
                    *     ENTRY: (X)=16-BIT DIVIDEND                  *
                    *            (A)=8-BIT DIVISOR                    *
                    *     EXIT:  (X)=QUOTIENT                         *
                    *            (A)=REMAINDER                        *
                    ***************************************************
0F71 34     12      DIV168  PSHS    X,A      DIVIDEND, DIVISOR
0F73 4F                     CLRA             CLEAR 1/2 OF DIVIDEND
0F74 E6     61              LDB     +1,S     GET MSB OF DIVIDEND
0F76 8D     0B              BSR     DIVIDE   DO 8 DIVIDES
0F78 E7     61              STB     +1,S     REPLACE 1ST 1/2
0F7A E6     62              LDB     +2,S     GET LSB OF DIVIDEND
0F7C 8D     05              BSR     DIVIDE   DO 8 DIVIDES
0F7E E7     62              STB     +2,S     REPLACE 2ND 1/2
0F80 35     14              PULS    B,X      DISCARD DIVISOR, GET Q
0F82 39             ENDEX   RTS              RETURN
0F83 8E     0008    DIVIDE  LDX     #8       SETUP COUNTER
0F86 58             DIV010  LSLB                SHIFT D LEFT ONE BIT
0F87 49                     ROLA
0F88 CA     01              ORB     #1          PRESET Q BIT TO 1
0F8A 24     04              BCC     DIV015      GO IF C=0
0F8C A0     62              SUBA    +2,S        SUBTRACT MUST GO
0F8E 20     08              BRA     DIV020      CONTINUE
0F90 A0     62      DIV015  SUBA    +2,S        DO SUBTRACT
0F92 24     04              BHS     DIV020      GO IF + OR 0
0F94 C4     FE              ANDB    #$FE        RESET Q BIT
0F96 AB     62              ADDA    +2,S        RESTORE
0F98 30     1F      DIV020  LEAX    -1,X        DECREMENT COUNT
0F9A 26     EA              BNE     DIV010      GO IF NOT 0
0F9C 39                     RTS              RETURN TO CALLING PROG
```

Figure 17-4. Signed Divide Program

```
                0000       00650              END
00000 TOTAL ERRORS

DIV010  0F86
DIV015  0F90
DIV020  0F98
DIV168  0F71
DIVDND  0F6E
DIVIDE  0F83
DIVSOR  0F70
ENDEX   0F82
LOOP    0F6B
SDI010  0F51
SDI020  0F59
SDIV    0F3E
```

Figure 17-4 continued

The trick here is to get the sign of the result. The sign of a multiply or a divide is the exclusive OR of the two operands:

```
+ times a + = +    0 EOR 0 = 0
+ times a - = -    0 EOR 1 = 1
- times a - = +    1 EOR 1 = 0
(same for divides)
```

The EOR is taken and saved in the stack. Bit 7 of the result value is the sign, 0 (+) or 1 (-). After the operands have been converted to their absolute values, the sign result is PULSed from the stack and used to convert the result of the divide to the proper sign.

The operands initially should be 16-bit or 8-bit two's complement numbers. The results will also be two's complement numbers. Can you detect a case in which the DIVIDE will not work properly?

If -32768 ($8000) is divided by -1 ($FF), the result of 32,768 is too large to be held in 16 bits. The largest positive 16-bit **signed** number is 32,767, or $7FFF.

Review

To review what we've learned in this chapter:

- Software divides generally use a "restoring" division technique of shift, subtract, and possible restore
- Numbers can be "scaled up" for multiplies and divides
- Signed multiplies and divides may be done by converting to absolute values, performing the operation, and converting the result to proper sign
- Exclusive ORing two operands for a multiply or divide will give the proper result sign in the sign bit of the result of the EOR

For Further Study

KEY CHART — CHAPTER 18

INSTRUCTIONS

~~*ABX*~~	~~*LBHS*~~	~~*CLRB*~~	~~*LDS*~~	~~*ROL*~~
~~*ADCA*~~	~~*BITA*~~	~~*CLR*~~	~~*LDU*~~	~~*RORA*~~
~~*ADCB*~~	~~*BITB*~~	~~*CMPA*~~	~~*LDX*~~	~~*RORB*~~
~~*ADDA*~~	~~*BLE*~~	~~*CMPB*~~	~~*LDY*~~	~~*ROR*~~
~~*ADDB*~~	~~*LBLE*~~	~~*CMPD*~~	~~*LEAS*~~	~~*RTI*~~
~~*ADDD*~~	~~*BLO*~~	~~*CMPS*~~	~~*LEAU*~~	~~*RTS*~~
~~*ANDA*~~	~~*LBLO*~~	~~*CMPU*~~	~~*LEAX*~~	~~*SBCA*~~
~~*ANDB*~~	~~*BLS*~~	~~*CMPX*~~	~~*LEAY*~~	~~*SBCB*~~
~~*ANDCC*~~	~~*LBLS*~~	~~*CMPY*~~	~~*LSLA*~~	~~*SEX*~~
~~*ASLA*~~	~~*BLT*~~	~~*COMA*~~	~~*LSLB*~~	~~*STA*~~
~~*ASLB*~~	~~*LBLT*~~	~~*COMB*~~	~~*LSL*~~	~~*STB*~~
~~*ASL*~~	~~*BMI*~~	~~*COM*~~	~~*LSRA*~~	~~*STD*~~
~~*ASRA*~~	~~*LBMI*~~	**CWAI**	~~*LSRB*~~	~~*STS*~~
~~*ASRB*~~	~~*BNE*~~	**DAA**	~~*LSR*~~	~~*STU*~~
~~*ASR*~~	~~*LBNE*~~	~~*DECA*~~	~~*MUL*~~	~~*STX*~~
~~*BCC*~~	~~*BPL*~~	~~*DECB*~~	~~*NEGA*~~	~~*STY*~~
~~*LBCC*~~	~~*LBPL*~~	~~*DEC*~~	~~*NEGB*~~	~~*SUBA*~~
~~*BCS*~~	~~*BRA*~~	~~*EORA*~~	~~*NEG*~~	~~*SUBB*~~
~~*LBCS*~~	~~*LBRA*~~	~~*EORB*~~	~~*NOP*~~	~~*SUBD*~~
~~*BEQ*~~	~~*BRN*~~	~~*EXG*~~	~~*ORA*~~	**SWI**
~~*LBEQ*~~	~~*LBRN*~~	~~*INCA*~~	~~*ORB*~~	**SWI2**
~~*BGE*~~	~~*BSR*~~	~~*INCB*~~	~~*ORCC*~~	**SWI3**
~~*LBGE*~~	~~*LBSR*~~	~~*INC*~~	~~*PSHS*~~	**SYNC**
~~*BGT*~~	~~*BVC*~~	~~*JMP*~~	**PSHU**	~~*TFR*~~
~~*LBGT*~~	~~*LBVC*~~	~~*JSR*~~	~~*PULS*~~	~~*TSTA*~~
~~*BHI*~~	~~*BVS*~~	~~*LDA*~~	**PULU**	~~*TSTB*~~
~~*LBHI*~~	~~*LBVS*~~	~~*LDB*~~	~~*ROLA*~~	~~*TST*~~
~~*BHS*~~	~~*CLRA*~~	~~*LDD*~~	~~*ROLB*~~	

EDTASM+ EDITOR COMMANDS

~~*A(SSEMBLE)*~~	~~*I(NSERT)*~~	~~*R(EPLACE)*~~
~~*C(OPY)*~~	~~*L(OAD)*~~	~~*T(HARDCOPY)*~~
~~*D(ELETE)*~~	~~*M(OVE)*~~	~~*V(ERIFY)*~~
~~*E(DIT)*~~	~~*N(UMBER)*~~	~~*W(RITE)*~~
~~*F(IND)*~~	~~*P(RINT)*~~	~~*Z(BUG)*~~
~~*H(ARDCOPY)*~~	~~*Q(UIT)*~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	/NO NO OBJECT
~~*/IM IN MEMORY ASSEMBLY*~~	~~*/NS NO SYMBOL TABL*~~
~~*/LP LINE PRINTER*~~	/SS SHORT SCREEN
/MO MANUAL ORIGIN	~~*/WE WAIT ON ERROR*~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~*A(SCII) DISPLAY*~~	~~*T DISPLAY BLOCK*~~
~~*B(YTE) MODE*~~	~~*T H HARDCOPY BLOC*~~
~~*C(ONTINUE)*~~	~~*U MOVE BLOCK*~~
~~*D(ISPLAY)*~~	~~*V (ERIFY) BLOCK*~~
~~*E(DITOR)*~~	~~*W(ORD) MODE*~~
~~*G(O)*~~	~~*X BREAKPOINT*~~
~~*H(ALF) SYMBOLIC*~~	~~*Y (ANK) BREAKPOINT*~~
~~*I(NPUT) BASE*~~	~~*↑ EXAMINE PRECEDIN*~~
~~*L(OAD) ML FILE*~~	~~*↓ EXAMINE NEXT*~~
~~*M(NEMONIC) MODE*~~	→ BRANCH INDIRECT
~~*N(UMERIC) MODE*~~	~~*; FORCE NUMERIC*~~
~~*O(OUTPUT) BASE*~~	~~*+ FORCE NUMERIC,BY*~~
~~*P SAVE ML ON TAPE*~~	~~*: FORCE FLAGS*~~
~~*R(EGISTER) DISPLAY*~~	~~*/ EXAMINE*~~
~~*S(YMBOLIC DISPLAY)*~~	~~*, SINGLE STEP*~~

ADDRESSING MODES

~~*INHERENT*~~
~~*DIRECT*~~
~~*EXTENDED*~~
~~*IMMEDIATE*~~
~~*SIMPLE INDEXED*~~
~~*RELATIVE*~~
~~*DISPLACEMENT INDEXED*~~
~~*AUTO INCREMENT/DECREMENT*~~
INDIRECT
SOPHISTICATED

GENERAL TOPICS

~~*CPU REGISTERS*~~	~~*SUBROUTINES*~~
~~*DATA TO REGISTERS*~~	~~*STACK OPERATIONS*~~
~~*LOADING AND STORING*~~	~~*ROTATES, SHIFTS*~~
~~*ADDITION AND SUBTRACTION*~~	~~*MULTIPLES*~~
~~*CONDITION CODES*~~	~~*DIVIDES*~~
~~*SYMBOLIC ADDRESSING*~~	**DECIMAL ARITHMETIC**
~~*JUMPS, BRANCHES*~~	BASIC INTERFACING
~~*RELATIVE BRANCHES*~~	PASSING PARAMETERS
~~*INCREMENTS/DECREMENTS*~~	VARPTR USE
~~*COMPLEMENTS*~~	ROM SUBROUTINES
~~*LOGICAL OPERATIONS*~~	OTHER ADDRESSING
~~*MULTIPLE PRECISION*~~	GRAPHICS
~~*DATA VALUES*~~	SOUND
~~*INDEXING*~~	LARGER PROGRAMS
~~*INDEXING WITH X,Y*~~	
~~*SORTING*~~	

PSEUDO OPS

EQU	ORG
~~*FCB*~~	**RMB**
~~*FCC*~~	SET
~~*FDB*~~	~~*SETDP*~~

Bold Type = Present Chapter
Regular Type = Future Chapters
~~*Italic Type*~~ *= Past Chapters*

Chapter 18
Decimal Arithmetic and Miscellaneous Instructions

Decimal arithmetic allows us to do arithmetic not in binary, but in "binary-coded-decimal," a way of representing decimal digits in 4 bits. The DAA instruction adjusts binary results to bcd form. In addition to looking at decimal operations here, we'll pick up some of the miscellaneous instructions that have "slipped through the crack."

The Decimal Instruction

The additions and subtractions we've been discussing in previous chapters have all been **binary** adds and subtracts. However, the 6809 has the capability of adding numbers in **decimal** format.

Binary-coded-decimal data represents decimal digits of 0 through 9 in groups of 4 bits:

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

The binary-coded-decimal values of 1010 through 1111 are not allowed. In this method of representation, each binary byte holds 2 bcd digits. The four binary bytes

00010010 00110100 01010110 01111001,

for example, would represent the 8 bcd digits 12345679.

By using a special instruction, the DAA, we can add and subtract bcd digits and have them come out properly.

Look at the following program.

```
                      00100 * BCD OPERATIONS
0A17  CE    0A3E      00110 BCDST    LDU      #RESULT+4    POINT TO LS BYTE+1
0A1A  8E    0A36      00120          LDX      #BCDOP2      POINT TO BCDOP1+4
0A1D  108E  0A3A      00130          LDY      #RESULT      POINT TO BCDOP2+4
0A21  1C    FE        00140          ANDCC    #$FE         RESET CARRY
0A23  C6    04        00150          LDB      #4           ITERATION COUNT
0A25  A6    82        00160 BCD010   LDA      ,-X            GET BYTE
0A27  A9    A2        00170          ADCA     ,-Y            ADD OP2 BYTE
0A29  19              00180          DAA                     DECIMAL ADJUST
0A2A  A7    C2        00190          STA      ,-U            STORE RESULT
0A2C  5A              00200          DECB                    DECR ITERATION CNT
0A2D  26    F6        00210          BNE      BCD010         GO IF NOT 4
0A2F  7E    0A2F      00220 LOOP     JMP      LOOP         LOOP HERE
0A32                  00230 BCDOP1   RMB      4            OP 1
0A36                  00240 BCDOP2   RMB      4            OP 2
0A3A                  00250 RESULT   RMB      4            RESULT
```

Figure 18-1. BCD Operations Program

```
               0000        00260            END
00000 TOTAL ERRORS

BCD010   0A25
BCDOP1   0A32
BCDOP2   0A36
BCDST    0A17
LOOP     0A2F
RESULT   0A3A
```

Figure 18-1 continued

This program takes two 4-byte operands and adds them together in a multiple-precision operation. The first 4-byte operand is located in locations BCDOP1 through BCDOP1+3. The second is located in locations BCDOP2 through BCDOP2+3. The bcd result is stored in locations RESULT through RESULT+3.

All operands are treated as 32-bit numbers, with the most significant byte on the left and the least significant byte on the right.

If you want to run the program, assemble, breakpoint at LOOP, and store some typical operands in BCDOP1 and BCDOP2. Here are some examples:

Operand 1: $12, $34, $56, $78

Operand 2: $56, $78, $91, $23

Execute from BCDST and look at the results after the breakpoint is reached.

Hints and Kinks 18-1
Using Auto Decrement

We haven't used auto **decrement** before this program, and we'd like to remind you that the decrement is done **before** the instruction execution. Here we've loaded U, X, and Y with one more than the last location of the operands to compensate for the way auto decrement works.

The result of a binary add is predictable and not too hard to figure out, even though we are working with 32-bit numbers:

```
 $12345678
+$56789123
 $68ACE79B
```

What does this number represent in binary? Some ungodly number, no doubt...That's not the point — look at the result of the **bcd** add in the RESULT locations.

The result of the bcd add was:

```
 $12345678
+$56789123
 $69134801
```

In fact, the bcd add enables us to treat the two operands as if they were decimal numbers, and not binary. The result was the same as if we had added the two numbers with pencil and paper.

The decimal adjust accumulator (DAA) takes each binary result and converts it to a bcd-format number. It adjusts the result from a binary result to a bcd result, eliminating the invalid bcd 4-bit groups of 1010, 1011, 1100, 1101, 1110, and 1111.

Try some other operands, and you'll see the difference. Of course, you must start off with valid BCD numbers in both of the operands, numbers that have the valid bcd digits of 0000 (0) through 1001 (9) in each digit position.

The DAA automatically handles "carries" also, so that you can work with "multiple-precision" bcd operands, just as you did with multiple-precision binary operands.

BCD numbers take more space, as you can see from the results. Conversions between ASCII and bcd are somewhat simpler, however. To convert from an ASCII character of $30 through $39, representing "0" through "9," all you have to do is something like:

```
CONVRT BSR   GETCHR    GET CHARACTER
       SUB   #$30      CONVERT TO BCD 0-9
       STA   ,X        STORE
```

The SUB above converts the ASCII $30 through $39 to $00 through $09.

Hints and Kinks 18-2
Efficiency of Storage Using BCD

Although BCD data is easy to decipher (you can see immediately that 10010011 is 1001,0011 or 93 bcd), it is much less efficient in storage than binary. You can store up to 4,295,000,000 in 32 bits in binary, while you can store only 99,999,999 in the same 32 bits in bcd!

The RMB Pseudo Op

We used an RMB, or "Reserve Memory Bytes" pseudo op in the code above. The RMB simply sets aside n bytes of storage; we could have specified something like

```
BCDOP1  FCB   0
        FCB   0
        FCB   0
        FCB   0
```

or

```
BCDOP1  FDB   0
        FDB   0
```

but the RMB is handy for reserving the space and marking the area with a label without having to enter a source line for every one or two locations. Note that **no** data, not even zeroes, is stored in the area reserved. It's similar to reserving space in a BASIC array with DIM.

Using the U Stack

Did you wonder about the U register above? The U register defines a "User" stack, which is similar to the S stack area. The big difference is that the S stack area is used automatically in BSRs, JSRs, and LBSRs, and in storage of interrupt addresses. The U stack is user designated and can be used in PSHU and PULU operations, and also as an index register!

We've used it as an index register above in lieu of using X or Y, which were dedicated to pointing to the operands for the bcd operations. When used this way, U is initialized just as an X or Y index register, and then used in the ,U format in Loads, Stores, and other instructions. Think of U as another index register in operations such as the one above.

The U stack can be used for storage of data by PSHUs and PULUs the same way as we used PSHSes and PULSes in previous programs (that's easy for you to say . . .). However, the U register must be loaded with the U stack area **plus 1** before any U stack operations can be done.

```
LDU   #$3000     SET UP U STACK
```

Where can this U stack be? Anywhere you want. There are no restrictions to the location of the U stack or the size of the U stack.

Once an LDU has been done, you can use the U stack for indexing, as we saw in the BCDST program, or for PULU or PSHU operations with impunity.

The NOP Instruction

What is a NOP? A NOP is just what it says, a "no operation." A NOP is used to delete instructions by substituting a NOP opcode for all bytes of the instruction.

Suppose that you had the following code:

```
LDA   ,X       GET BYTE
LEAX  1,X      BUMP PNTR
LEAX  1,X      BUMP AGAIN
```

and you found out that the second LEAX 1,X was not needed. You could effectively delete the LEAX 1,X without reassembling by replacing the $30, $01 bytes of the LEAX 1,X by NOP codes of $12 and $12. The NOP does not affect any registers or any condition code settings.

Hints and Kinks 18-3
The CWAI Instruction

The CWAI mnemonic stands for "Clear and Wait for Interrupt." This instruction is a "preparation" for an expected interrupt. It saves the "environment" in the S stack and then waits for an interrupt. It's used in an "interrupt-driven" assembly-language program which is beyond the scope of this book.

Hints and Kinks 18-4
The SYNC Instruction

The SYNC instruction is used in "interrupt-driven" applications programs for high-speed input/output. Again, this process is beyond the scope of this book. (What do you want for 95 cents, anyway?. . .)

Hints and Kinks 18-5
SWI, SWI2, and SWI3

These three instructions provide "software" interrupts. Software interrupts are used in "queueing" tasks in a interrupt-driven environment. We'll cover these topics in the 32nd book of this series, available in late 1987.

How to Use 6809 Instructions

At this point we've covered all 6809 instructions and many addressing modes. We'll "fill in the gaps" on the addressing modes shortly. Remember one guiding rule in working with the instruction set: **There is not necessarily a right way to do things.** Many times the same program can literally be implemented hundreds of different ways. Feel free to experiment and try new approaches. You can't go too far wrong. Assembly-language is so fast that things will still move swiftly.

Review

To review what we've learned here:

- BCD operands utilize the binary-coded-decimal digits of 0000 through 1001 to represent decimal digits of 0 through 9

- The DAA does a decimal adjust of the A register after an add
- The RMB pseudo op reserves memory bytes, but does not fill the reserved area with data
- The U stack can be used in identical fashion to the S stack for indexing and PULU and PSHU operations
- NOP is a "do nothing" instruction primarily used for patching

For Further Study

KEY CHART — CHAPTER 19

INSTRUCTIONS

-	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~A~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~B~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~A~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~B~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~D~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~A~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~B~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~CC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~A~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~B~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
.	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~A~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~B~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
.	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~.~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~C~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~CS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~D~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~EQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~E~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~GE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~T~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~GT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~I~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~HI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~S~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~HERENT~~
~~RECT~~
~~TENDED~~
~~MEDIATE~~
~~MPLE INDEXED~~
~~LATIVE~~
~~PLACEMENT INDEXED~~
~~TO INCREMENT/DECREMENT~~
DIRECT
PHISTICATED

PSEUDO OPS

U	**ORG**
~~B~~	~~RMB~~
~~C~~	SET
~~B~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

/AO ABSOLUTE ORIGIN	**/NO NO OBJECT**
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	**/SS SHORT SCREEN**
/MO MANUAL ORIGIN	~~/WE WAIT ON ERRORS~~
/NL NO LISTING	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	→ BRANCH INDIRECT
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	**BASIC INTERFACING**
~~RELATIVE BRANCHES~~	PASSING PARAMETERS
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

d Type - Present Chapter
gular Type = Future Chapters
~~lic Type~~ = *Past Chapters*

Chapter 19
Program Origin and Interfacing Assembly Language to BASIC

In this chapter we'll see how assembly-language programs can be linked to BASIC. This is a fairly simple process. It involves assembling the program at the right spot in memory, defining that location to the BASIC interpreter, and transferring control by something similar to a "BSR." For short programs, the assembly-language code may be converted to BASIC DATA values and incorporated into the BASIC program.

One of the best ways to learn assembly language is to interface it with BASIC in short, high-speed subroutines that complement the flexibility of BASIC. In the next few chapters we'll show you how to do that.

Memory Map

First we'll have to get a clear idea of where we can put assembly-language subroutines. Look at Figure 19-1. It shows the general memory layout of the Color Computer. Some of it may be familiar to you.

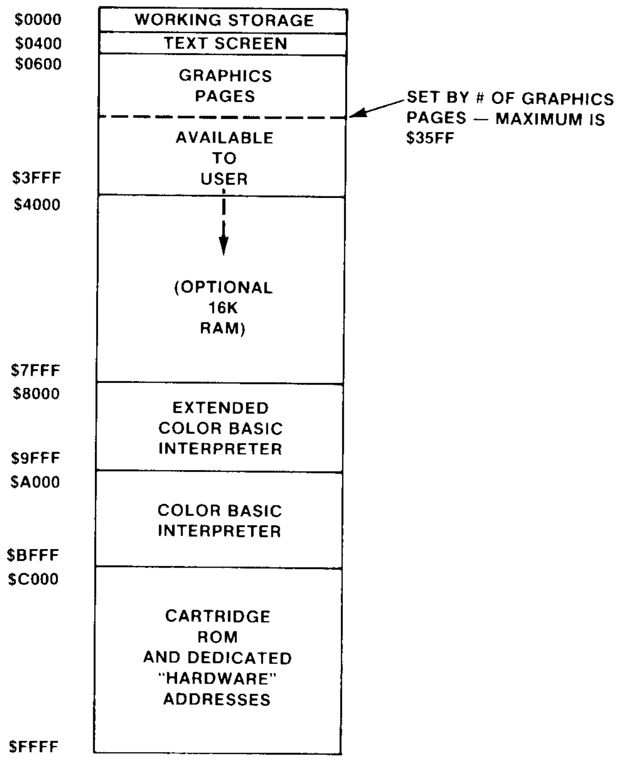


Figure 19-1. Memory Layout of Color Computer

19 Program Origin and Interfacing Assembly Language to BASIC

The Color BASIC and Extended Color BASIC interpreter is "burned into" ROM memory locations $A000 through $BFFF and $8000 through $9FFF, respectively. This, and all locations from $8000 through $FFFF, are the "ROM" or "Read Only Memory" portion of the 64K (65,536) bytes of memory available to the system.

The memory addresses from $0000 through $03FF, the first two 256-byte pages of RAM (random-access-memory) are used to hold system variables and as "working storage" for the BASIC interpreter. These locations should never be used for assembly-language programs.

The text screen, as we've seen in previous chapters, is located starting at $0400. The text screen ends at $05FF. The text screen is actually a part of normal RAM memory. In general these locations can be addressed as normal memory locations. To write an ASCII "A" at the upper left-hand corner of the screen, for example, you'd simply do something like this:

```
LDA   #65       ASCII A
STA   $0400     STORE
```

The area from $0600 on is used for 1 to 8 graphics "pages." The maximum number of graphics pages, 8, would use locations $0600 through $35FF. Each graphics page is 1536 bytes long.

If you are running a pure assembly-language program, without using any BASIC interfacing, then you will have all of the RAM from $0000 through top of memory for your use.

If you are running a combination BASIC and assembly-language program, you'll find that parts of the RAM area from the end of the graphics screens are taken up by BASIC program lines, by simple variable storage, by array storage, by the string storage area, and by the basic stack. The general scheme is shown in Figure 19-2.

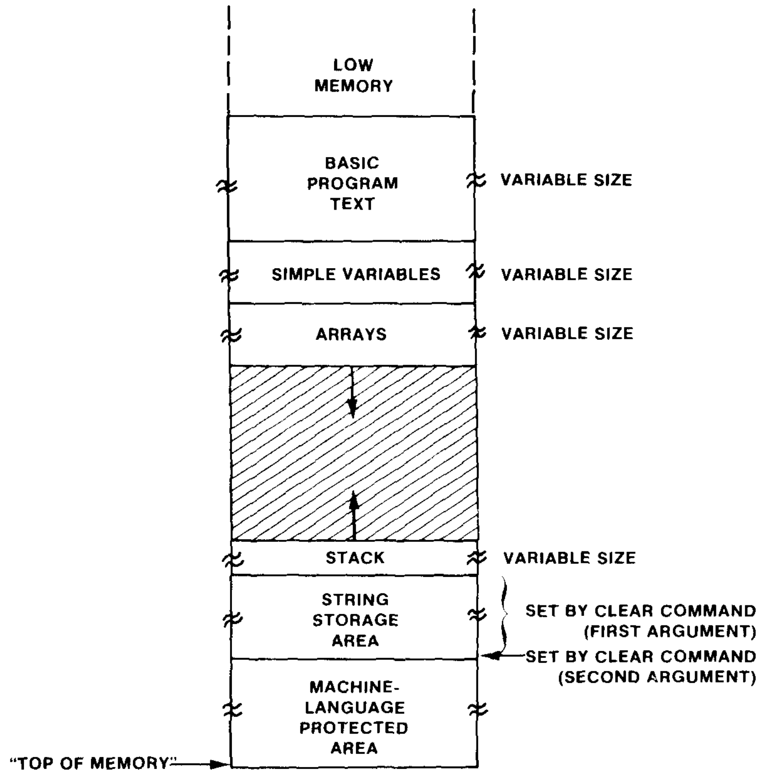


Figure 19-2. RAM Storage

The best place to put a short assembly-language program is as close to the top of memory as possible. You may protect memory used for assembly-language programs by entering a memory protect value by CLEAR XXX,16127 when you first load BASIC. The "XXX" portion of the CLEAR allocates XXX bytes for string storage, and the next address is one less than the start of the area to be protected. A typical XXX value would be 200.

If you enter CLEAR 200,16127, for example, you'd protect all of memory from 16128 ($3F00) on. BASIC would not use any of that area, and you could use it for assembly-language programs, or for any other operations.

The ORG (Origin) Command and /AO

All of the programs we've worked with up to this point have started at the end of the text buffer/symbol table area in EDTASM+. In other words, EDTASM+ assembles the object code starting at the first location after the text and symbol table. However, we can use an optional assembler pseudo-op to determine the object start. It is called ORG, for ORiGin, and has the format

```
          ORG    $XXXX,
```

where $XXXX is a hexadecimal starting location (or decimal).

The origin may be anywhere in RAM that you'd like, as long as it is greater than the end of the text/symbol table area, and less than the top of memory.

A practical starting point would be somewhere around $3F00. This would be out of the text area for most small programs, and far enough down so that it would not interfere with the "symbol table" and other assembler areas. However, this starting point would only give you 256 bytes of program area. This will be sufficient for the programs we're going to use here, but you might want to use a lower protected address for your own larger programs.

To show you how ORG works, look at the following program.

```
* SAMPLE USE OF ORG
          ORG    $3F00     ASSEMBLE AT $3F00
START     LDA    NEXT      LOAD A
          BRA    CONT      JUMP AROUND DATA
NEXT      FCB    23        DATA
CONT      JMP    CONT
          END              END
```

Assemble the program by A/AO/IM. Note that the object code starts at location $3F00. Jot down the object code or do an A/LP listing of the object code by assembling with your system printer.

Now change the Origin to $3E00 and reassemble by A/AO/IM:

```
* SAMPLE USE OF ORG
          ORG    $3E00     ASSEMBLE AT $3E00
START     LDA    NEXT      LOAD A
          BRA    CONT      JUMP AROUND DATA
NEXT      FCB    23        DATA
CONT      JMP    CONT
          END              END
```

Did you notice any difference in the object code?

Hints and Kinks 19-1
The /MO Assembler Option

/MO stands for "manual origin." It allows you to define the start address of the assembled program and the start address of the edit buffer/symbol table. It shouldn't be used (because it's too much trouble) unless you specifically have to redefine the memory layout. Why would you want to redefine the memory layout? You might have code that is not relocatable and is designed to run at location $0900, right in the middle of the edit buffer/symbol table. /MO gives you some control over the assembly for infrequent odd configurations.

There are two variables in EDTASM+, USRORG ($FD) and BEGTMP ($FF), which define the start of the object code and start of the edit buffer/symbol table, respectively. Before running in the /MO mode, redefine these locations as follows:

Using ZBUG, set locations $FD and $FE (USRORG) to the object code start. Setting $FD/$FE to $3000, for example, will result in object code being assembled in memory starting at location $3000.

Using ZBUG, set location $FF (BEGTMP) to a $07 through $7F. This variable is similar to the DP register in that it defines the start of a 256-byte memory page that will be the start of the edit buffer/symbol table. Setting location $FF to $15, for example, would set the edit buffer/symbol table start to $1500. The starting location must be less than the USRORG value.

Again, don't use this mode unless you must, only because it's some trouble to set it up.

Relocatability

The assembly at ORG $3F00, had

```
3F00   B6 3F05   START   LDA   NEXT
3F03   20 01             BRA   CONT
3F05   17        NEXT    FCB   23
```

The address of 3F05 in the LDA instruction refers to the "absolute" location of $3F05.

The assembly at ORG $3E00 had:

```
3E00   B6 3E05   START   LDA   NEXT
3E03   20 01             BRA   CONT
3E05   17        NEXT    FCB   23
```

Even though both LDAs referred to a location 6 bytes away, they used different addresses.

Absolute addresses in Loads, Stores, and other instructions are the reason that object code cannot simply be moved to a new location and execute properly. A reassembly must be done.

Some instructions are "relocatable," though. A TFR A,B will work anywhere it is located. A relative jump also works anywhere, as it contains no absolute address, but simply a displacement from the current program counter. (Look at the BRA instruction for the two assemblies.)

How to Make All Code Relocatable

In fact, all machine-language code on the 6809 can be made "position independent" or "relocatable," but at the expense of using a more complicated addressing mode called "PC relative." We'll discuss PC relative addressing in a later chapter. For now, though, we'll assume that some code in the examples we're using will not be relocatable.

Transferring Control to an Assembly-Language Program

Just how do you transfer control to an assembly-language program from BASIC? There are three steps:

1. Loading the object code of the assembly-language program into RAM
2. Defining where the object code is to the BASIC interpreter
3. Transferring control to the assembly-languge program by a USR call in BASIC.

To show you how this process works, let's use the following program. It is a very simple program to clear the video display. (Never mind that there is a CLS command in BASIC; this time we want to do it ourselves!)

```
                   00100 * CLEAR SCREEN PROGRAM
3F00               00110        ORG   $3F00   ORIGIN
3F00 86   20       00120 CLRSCN LDA   #$20    BLANK
3F02 8E   0400     00130        LDX   #$400   START OF SCREEN
3F05 A7   80       00140 CLR010 STA   ,X+       STORE BLANK
3F07 8C   0600     00150        CMPX  #$600     TEST END
3F0A 26   F9       00160        BNE   CLR010    GO IF NOT
3F0C 39            00170        RTS           RETURN
          0000     00180        END
00000 TOTAL ERRORS

CLR010  3F05
CLRSCN  3F00
```

Figure 19-3. Clear Screen Program

If you assemble the program above (use A/IM/AO), you should get an assembly similar to Figure 19-3. The important point is that the object code should be identical.

Hints and Kinks 19-2
The Clear Screen Program

As it turns out, this program is relocatable. If you look at the instructions, none of them specify absolute addresses **within** the program. There are absolute addresses, all right — the screen start area is always at $400, and one more than the screen end is at $600 — but these addresses never change in any program and are hence relocatable. The BNE is, of course, relative and relocatable. You might want to modify the BASIC program to move the code somewhere else. If you do, be certain to protect the new area with a CLEAR statement.

Loading the Object

We could assemble this code by

```
*A NAME
```

and get the object output on cassette tape. The program could then be loaded by doing a

```
OK
CLOADM "NAME"
```

from BASIC, which would load in the machine-language file created by EDTASM+. An

```
EXEC
```

would then start the program.

However, we want to interface and control this program with BASIC, and must therefore follow a different procedure. We'll load the object by using the POKE statement in BASIC.

In case you're not familiar with the POKE, it works like this:

```
POKE      16128,21      'POKE 21 INTO 16128
```

The POKE statement above stores a decimal 21 into RAM location 16128 decimal, which corresponds to $3F00.

There's only one problem with the POKE. It works with **decimal** values in Color BASIC, which means that we have to convert the **hexadecimal** values from the assembly-language listing into decimal before we can use the POKE! Extended Color BASIC users, however, can use the BASIC form

```
POKE &H3F00,&H15
```

to specify hexadecimal addresses directly. Just for consistency's sake, we'll use the decimal form in the following examples.

You won't find converting from hexadecimal to decimal too much of a chore as we've given you an equivalence table in Appendix IV. Also, for every program in the book we'll give you the actual decimal values along with the hexadecimal values.

The hexadecimal values for the program (from the listing) are $86, $20, $8E, $04, $00, $A7, $80, $8C, $06, $00, $26, $F9, $39.

The corresponding decimal values (from Appendix IV) are 134, 32, 142, 4, 0, 167, 128, 140, 6, 0, 38, 249, 57.

How do we get these values into RAM? An easy way in BASIC is is to put the values into DATA statements, and then POKE them into RAM with a short loop:

```
100 DATA 134, 32, 142, 4, 0, 167, 128, 140, 6, 0,
    38, 249, 57

110 FOR I=16128 TO 16128+12

120 READ A

130 POKE I,A

140 NEXT I
```

In this loop a READ command gets the values from the DATA list. The POKE then POKEs the value into the current memory location (don't forget that 16128 corresponds to hexadecimal $3F00, the ORG point of the program).

The "12" in the FOR...TO statement corresponds to 1 less than the size of the program. If we had a program of 100 bytes, we'd use "16128+99."

Hints and Kinks 19-3
DATA Values in Hex

If you have Extended Color or Disk BASIC, you can use the "&H" prefix and use hex values for every DATA value. This makes embedded machine-language code in BASIC much less laborious (I just spent one hour looking for a conversion error in a DATA value!).

We'll assume that we've run the BASIC code above and that the object program has been loaded by the POKEs. At this point RAM locations $3F00 through $3F0C contain the CLRSCN program in "machine language." Nothing mysterious here; we've been doing the same thing in our EDTASM+ programs except that we weren't going to run BASIC along with the program. The next step is to tell the BASIC interpreter where the program is.

Defining Where the Object Is to BASIC

If you are running Color BASIC, the following statements will tell the BASIC interpreter where CLRSCN is located:

```
150 POKE 275,63: POKE 276,0
```

These statements store 16128 into locations 275 and 276. Locations 275 and 276 are simply a BASIC "variable" that defines where the machine-language program is located.

If you are running Extended Color BASIC or Disk BASIC, the following statement will tell BASIC where CLRSCN is located:

```
150 DEFUSR0=&H3F00
```

This statement assigns the code of "0" to the CLRSCN location of $3F00. We could just as well have used DEFUSR3=&H3F00, assigning a code of 3.

Transferring Control to the Machine-Language Code

We're all set up now to transfer control to CLRSCN. We'll do it by a USR call. The BASIC USR call gets the location from either the 275/276 variable or from the DEFUSR variable and simply does a BSR. That pushes the return address to the BASIC interpreter in the stack and saves the return point. The last statement in our CLRSCN program is an RTS, which pops the stack and causes a return to the BASIC interpreter.

If you have Color BASIC, this statement will do the job:

```
160 A=USR(0)
```

If you have Extended Color or Disk BASIC, this statement is the one to use:

```
160 A=USR0(0)
```

Executing CLRSCN

We've combined all of the statements above into a BASIC program that will:

1. Move the CLRSCN values from DATA statements into the $3F00 area
2. Define the location to BASIC
3. Transfer control to CLRSCN

If you execute the program below, you should see the screen clear in a flash. Load the program by going to BASIC and doing a CLOAD "CLRSCN" from cassette or a LOAD "CLRSCN" from disk. Before you do, however

PROTECT MEMORY BY ENTERING CLEAR 200,16127

Color BASIC users: Delete statements 153 and 163: Extended Color or Disk BASIC users: Delete statements 151 and 161.

RUN the BASIC program.

After the screen clears, you can interrupt the BASIC program by pressing BREAK.

```
100 DATA 134,32,142,4,0,167,128,140,6,0,38,249,57
110 FOR I=16128 TO 16128+12
120 READ A
130 POKE I,A
140 NEXT I
151 POKE 275,63: POKE 276,0  '(COLOR BASIC)
153 DEFUSR0=&H3F00  '(EXTENDED COLOR OR DISK BASIC)
161 A=USR(0)  '(COLOR BASIC)
163 A=USR0(0) '(EXTENDED COLOR OR DISK BASIC)
170 GOTO 170
```

Figure 19-4. Clear Screen in BASIC Program

Hints and Kinks 19-4
More EDTASM+ Assembly Options

Time we pulled together loose ends and at least mentioned the other assembler "switch" options.

The /NO option would be used for "no object" when you weren't assembling in memory. If the /NO option were not used, EDTASM+ would try to write out an object file to cassette tape instead of in memory. The /NO results in a display of the assembly output on the screen but no object either in memory or as a cassette file.

The /NL is used to inhibit the display of the assembler listing. This could be used for very long assemblies where even listing the EDTASM+ output on the screen took a considerable amount of time.

The /SS stands for "Short Screen." Using this option produces one line of location and contents, followed by a second line of source code. It makes the display easier to read.

Review

To review the considerable material we've covered here:

- Assembly-language programs can be anywhere in RAM when run without BASIC interface
- Assembly-language programs should be in high RAM when run with BASIC and this area should be protected by a CLEAR XXX,YYYYY where YYYYY is the start address of the program-1
- The ORG pseudo op establishes the assembly origin for a program
- LDs, STs, and other instructions may not be "relocatable"; they may contain absolute addresses that prevent them from running anywhere in memory
- Loading object code can be done by POKEing in BASIC after first converting hexadecimal values to decimal
- The location of the assembly-language program must be defined to the BASIC interpreter by storing the address in locations 275,276 (Color BASIC) or by using a DEFUSR statement (Extended Color and Disk BASIC)
- A USR call transfers control to the assembly-language program

For Further Study

BASIC DEFUSR in Extended and Disk BASIC (BASIC manual)

BASIC USR or USRn command (BASIC manual)

KEY CHART — CHAPTER 20

INSTRUCTIONS

~~ABX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~ADCA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~ADCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~ADDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~ADDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~ANDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~ANDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ANDCC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~ASLA~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~ASLB~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
~~ASL~~	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~ASRA~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~ASRB~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
~~ASR~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~BCC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BCS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~LBEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~BGE~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~LBGE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~BGT~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~LBGT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
~~DISPLACEMENT INDEXED~~
~~AUTO INCREMENT/DECREMENT~~
INDIRECT
SOPHISTICATED

PSEUDO OPS

EQU	~~ORG~~
~~FCB~~	~~RMB~~
~~FCC~~	SET
~~FDB~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABL~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERRORS~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOC~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDIN~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	→ BRANCH INDIRECT
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BY~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	ROTATES, SHIFTS
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	**PASSING PARAMETERS**
~~INCREMENTS/DECREMENTS~~	VARPTR USE
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Bold Type Present Chapter
Regular Type Future Chapters
~~*Italic Type*~~ *Past Chapters*

Chapter 20
Passing Parameters to BASIC Programs

In this chapter we'll discuss how to pass "parameters" between BASIC programs and assembly-language subroutines. BASIC has a built-in capability of allowing one 16-bit argument to be passed in either direction.

We covered quite a bit of ground in the previous lesson. Let's review what we did.

First of all we assembled an assembly-language program using EDTASM+. This program took the form of a "subroutine," which, as we know, is really any program that is terminated by an RTS, to return to a calling program.

The program was assembled at location $3F00 by using an ORG statement. We aimed for this area because we knew that we were going to be calling the program from BASIC and wanted to locate the program in high enough memory to protect it from overwriting by BASIC statements, variables, and other data.

After the assembly, we didn't load the program directly into RAM by loading a cassette object file, although we could have by using EDTASM+ assembly options and the CLOADM and EXEC BASIC commands.

Instead, we converted the hexadecimal values from the assembly listing into decimal values by using Appendix IV.

We then made up a BASIC program that consisted of three parts

1. A DATA statement, or several DATA statements that had all of the machine-language decimal values
2. A short FOR...TO loop to move these values from the DATA statements into RAM at the $3F00 area
3. A BASIC statement that defined where the assembly-language program was, either by a POKE 275,63:POKE 276,0 or by DEFUSR0=16128
4. A BASIC call to the assembly-language program by a USR or USR0 statement

We then executed the BASIC program. The program moved the machine-code values from the DATA statements into the $3F00 area. It then took the address of the assembly-language program from either 275/276 or the DEFUSR statement and transferred control by a simple BSR instruction, somewhere in the BASIC interpreter.

The CLRSCN program then executed without any BASIC interference. It's important to note that once the assembly-language program is entered, BASIC has no control over it! That's good and bad — if the program has errors in it, there's no easy way to recover, as it may have destroyed critical memory locations used by BASIC! On the other hand it lets us execute the assembly-language code very rapidly, to supplement BASIC processing.

20 Passing Parameters to BASIC Programs

The last instruction in the CLRSCN was an RTS. The RTS did not perform magic, but only pulled the return address from the stack and caused a return to the BASIC interpreter at some internal point. The "internal point," by the way, is a set of BASIC interpreter code that handles the USR call to assembly-language programs.

The stack used here is an internal BASIC stack, and we don't have to be concerned about establishing our own stack area. There's enough room in the BASIC stack for just about everything we'd want to do in simple assembly-language code. Besides that, if we used our own stack, we'd never be able to return to BASIC unless we carefully saved the return address by something like

```
PULS  X          GET RETURN
STX   RETURN     SAVE FOR RETURN
```

and then used the return address to return to BASIC at the end of the assembly-language program.

Passing a Parameter

It's easy to see how the DEFUSR works (or the POKE 275/276), but what about the USR call? We used the format

```
100  A=USR(0)        or        100  A=USR0(0)
```

The first term is for Color BASIC and the second for Extended Color BASIC or Disk BASIC.

What is the (0) and the A variable?

BASIC takes the value from inside the parentheses, assumed to be a 16-bit integer value, and stores it in a special variable inside BASIC. This value can be accessed by the assembly-language program.

For example, if we wanted to "pass" a value of 223 instead of 0, we'd say

```
100 A=USR(223)          or    100 A=USR0(223)
```

If we wanted to pass the value of a variable, we'd say

```
100 A=USR(B)            or    100 A=USR0(B)
```

We could even pass the value of an expression, as in

```
100 A=USR(ZZ/256)       or    100 A=USR0(ZZ/256)
```

The only requirement for the value is that it would have to be a BASIC "integer" value of -32768 to +32767. In fact though, we can fool the BASIC interpreter into accepting an absolute value of 0 through 65,535 by using the following rules:

1. If the value is less than 32768, use the value alone:

```
100 A=USR(30000)        or    100 A=USR0(30000)
```

2. If the value is equal to or greater than 32768, use this form

   100 A=USR(40000-65536)   or   100 A=USR0(40000-65536)

You've seen in previous programs why we want to pass "parameters" to subroutines. In the SCANTY subroutine of Chapter 15, for example, we passed a pointer to a table and the size of the table. Parameters make the subroutine more flexible and generalized.

The USR call, then, lets us pass one 16-bit value as a parameter to the assembly-language subroutine.

Passing a Parameter Back

What about going the other direction? Just as you might suspect, BASIC also allows us to pass a 16-bit value **back** from the assembly-language subroutine to BASIC. You might use the assembly-language code, for example, to scan a table for a certain value and pass back the location of the found value, if any.

The A variable in

100 A=USR(B)   or   100 A=USR0(B)

is set equal to the 16-bit value returned by the assembly-language subroutine. Of course, if the assembly-language subroutine doesn't need to return a value, then A is not used, and is a "dummy," just as the 0 value was in

100 A=USR(0)   or   100 A=USR0(0)

A Sample Parameter-Passing Subroutine

To show you how this works, and what we must do in the assembly-language subroutine, look at the following program.

```
3FØØ                 ØØ1ØØ          ORG     $3FØØ
                     ØØ11Ø *****************************************************
                     ØØ12Ø * SUBROUTINE TO FIND SQUARE ROOTS                    *
                     ØØ13Ø *      ENTRY: SQUARE IN 16 BITS                      *
                     ØØ14Ø *      EXIT+  SQUARE ROOT IN 16 BITS                 *
                     ØØ15Ø *****************************************************
3FØØ BD   B3ED       ØØ16Ø SQROOT   JSR     $B3ED    GET ARGUMENT
3FØ3 8E   FFFF       ØØ17Ø          LDX     #-1      INITIALIZE SQUARE RT
3FØ6 1Ø8E ØØØ1       ØØ18Ø          LDY     #+1      INITIALIZE ODD INTEGER
3FØA 34   2Ø         ØØ19Ø          PSHS    Y        PUT IN STACK
3FØC 3Ø   Ø1         ØØ2ØØ SQRØ1Ø   LEAX    1,X        SQUARE RT +1
3FØE 1ØAE E4         ØØ21Ø          LDY     ,S         ODD INTEGER
3F11 31   3E         ØØ22Ø          LEAY    -2,Y       -1, -3, ETC
3F13 1ØAF E4         ØØ23Ø          STY     ,S         BACK TO STACK
3F16 E3   E4         ØØ24Ø          ADDD    ,S         SUBTRACT
3F18 25   F2         ØØ25Ø          BCS     SQRØ1Ø     LOOP IF NOT MINUS
3F1A 1F   1Ø         ØØ26Ø          TFR     X,D      SQ ROOT NOW IN D
3F1C 32   62         ØØ27Ø          LEAS    +2,S     RESET STACK
3F1E BD   B4F4       ØØ28Ø          JSR     $B4F4    CONVERT
3F21 39              ØØ29Ø          RTS              RETURN
          ØØØØ       ØØ3ØØ          END
ØØØØØ TOTAL ERRORS

SQRØ1Ø   3FØC
SQROOT   3FØØ
```

Figure 20-1. Square Root Program

If you assemble the program, you'll get something close to Figure 20-1. The object code should be identical to that in the figure.

This program is a modification of the square root program found in an earlier chapter. The parameter on entry is a 16-bit square from 0 through 65,535 (unsigned). On exit, the square of 0 through 255 is found. The square is taken to the next lower integer for fractional squares.

Hints and Kinks 20-1 Stack Use in SQROOT

We've pulled some fancy shenanigans in this program with the S stack. (Just for perspective, what we've done here would be sneered at by many sociopathic programmers as trivial. . .)

First of all, we stored the "odd integer" in the stack by a PSHS. The S register now points to the last byte of the 2-byte odd integer. Don't forget that the S register is always decremented **before** the PSHS.

Now we've done an LDY ,S. This loads the 2-byte value from the stack, as the S register is used as an "index register" and it points to the last (most significant byte) of the odd integer value pushed. The least significant byte is one byte higher in the stack.

The odd integer value in Y is then decremented by 2 and stored back in the S stack.

Finally, the ADDD ,S adds the odd integer value in the stack to the contents of the D register.

During all of these operations the S register pointed to the last byte of the last data in the stack. This type of stack reference is not uncommon. If we had wanted to use a previous 16-bit value that was PSHSed in the stack, we would have retrieved it by +2,S; if there was still another previous PSHS of a 16-bit value, we would have used +4,S. The S register in these operations serves as a type of index register which points to the "base address" of the last byte pushed onto the stack.

Hints and Kinks 20-2 Relocatability of SQROOT

Again, as in the last chapter, SQROOT is relocatable. The JSR $B3ED and JSR $B4F4 instructions contain absolute adddresses, but these addresses are outside the program area and will never change. The remainder of the instructions are either "inherent addressing" types that contain no addresses, or relative branches. Again, you can move SQROOT anywhere you'd like as long as the area is protected.

If you compare the earlier version of the program with this one, you'll notice two additional instructions. The JSR $B3ED instruction at the very beginning is a JSR to a BASIC interpreter subroutine. The BASIC subroutine finds the argument from the USR call and puts it into the D register after first converting it to integer form. This is the way that BASIC passes that 16-bit value from the USR call. At the LDX #-1 instruction, therefore, we've got the argument from BASIC in D.

The JSR $B4F4 instruction jumps back to a BASIC routine that takes the contents of the D register, converts it to a "floating-point" format used for BASIC variables, and stores it into the variable used in the USR call before the equals sign. If we had

```
100 ZZ=USR(B)    or    100 ZZ=USR0(B)
```

for example, variable ZZ would contain the square after the return to BASIC. Note that if **no argument** is to be returned, an RTS instruction is used. Only if the argument in D is to be returned is the JSR $B4F4 used before the normal RTS.

It seems, then, that the entry to assembly language is with an optional argument in D and that the exit from assembly language is by an optional argument in D.

Running the Subroutine in BASIC

The code below shows the SQROOT subroutine converted to BASIC decimal values. (Extended Color BASIC users can use &HXX values directly in the DATA statements.)

```
100 REM SQUARE/SQUARE ROOT
105 CLEAR 200,16127
110 CLS
120 DATA 189,179,237,142,255,255,16,142,0,1,52,32,48,1
130 DATA 16,174,228,49,62,16,175,228,227,228,37,242
135 DATA 31,16,50,98,189,180,244,57
140 FOR I=16128 TO 16128+33
150 READ A
160 POKE I,A
170 NEXT I
181 POKE 275,63: POKE 276,0  '(COLOR BASIC)
183 DEFUSR0=&H3F00  '(EXTENDED COLOR AND DISK BASIC)
190 INPUT SQ
201 SR=USR(SQ)  '(COLOR BASIC)
203 SR=USR0(SQ) '(EXTENDED COLOR AND DISK BASIC)
210 PRINT "SQUARE=";SQ,"SQUARE RT=";SR
220 GOTO 190
```

Figure 20-2. Square Root in BASIC Program

The program should print the square roots of all numbers input. If the SQ (square) is greater than 32,767, however, you will have a problem. BASIC will accept the entry on the INPUT statement, but when it comes time to make the USR call, BASIC will see that the value is larger than the maximum possible for integer values of +32,767.

One way to fix this is by:

```
201 SQ=USR(SQ-65536)    '(COLOR BASIC)
203 SQ=USR0(SQ-65536)   '(EXTENDED COLOR AND
                          DISK BASIC)
```

This will work for all values greater than +32,767, but will not work for values of 0 through +32,767. What we really need is

```
199 SX=SQ:  IF SQ>32767  THEN SX=SX-65536
201 SR=USR(SX)   '(COLOR BASIC)
203 SR=USR0(SX)  '(EXTENDED COLOR AND DISK BASIC)
```

As SR is returned as a value from 0 to 255, there is no equivalent problem with it.

Review

To recap what we've learned in this chapter:

- It's not necessary to use your own stack when interfacing to BASIC; BASIC maintains its own
- The USR call passes the argument within parentheses in the USR( ) call if a "JSR $B3ED" is done in the assembly-language program
- The USR call returns the argument from the assembly-language program if a "JSR $B4F4 is done directly before the RET in the assembly-language program
- Variables are passed in the D register
- Variables must be 16-bit integer variables
- If variables are over 32,767, then the XXXXX-65536 form must be used to fool the BASIC interpreter

For Further Study

BASIC variables (BASIC manual)

KEY CHART — CHAPTER 21

INSTRUCTIONS

~~F~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~CA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~CB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~DA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~DB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~DD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~IA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~IB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ICC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~A~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~B~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
F	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~RA~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~RB~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
~~R~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~C~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~CC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~S~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~CS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~Q~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~EQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~E~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~GE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~T~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~GT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~I~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~HI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~IS~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~HERENT~~
~~RECT~~
~~TENDED~~
~~MEDIATE~~
~~MPLE INDEXED~~
~~LATIVE~~
~~SPLACEMENT INDEXED~~
~~ITO INCREMENT/DECREMENT~~
DIRECT
PHISTICATED

PSEUDO OPS

U	~~ORG~~
~~B~~	~~RMB~~
~~C~~	SET
~~B~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERRORS~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	→ BRANCH INDIRECT
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	+ ~~FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	: ~~FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	/ ~~EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	, ~~SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	~~PASSING PARAMETERS~~
~~INCREMENTS/DECREMENTS~~	**VARPTR USE**
~~COMPLEMENTS~~	ROM SUBROUTINES
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

ld Type – Present Chapter
gular Type – Future Chapters
~~lic Type~~ – *Past Chapters*

Chapter 21
VARPTR and Passing Multiple Arguments

Extended Color and Disk BASIC have the VARPTR command, which permits the user to find the address of a BASIC variable. This command can be used to pass the location of strings or arrays to an assembly-language subroutine. Another topic we'll discuss in this chapter is how to pass "multiple" arguments between BASIC and assembly-language subroutines, rather than just the one 16-bit argument allowed in the BASIC USR call.

The VARPTR Command

Those of you with Color BASIC can skip this section, as only Extended Color and Disk BASIC have the VARPTR function.

VARPTR lets us find the address of any variable in a BASIC program. This is important to assembly-language subroutines because it allows the assembly-language subroutine to access BASIC data such as arrays and strings.

The format of VARPTR is

```
        100 VARPTR(XX)
```

where XX is a variable name.

VARPTR can be used either with numeric variables, array variables, or with string variables.

Using VARPTR With Strings

When VARPTR is used with a string variable, it has the format of

```
        100 A=VARPTR(AA$)
```

where AA$ is any string variable name.

VARPTR will put the address of a string "descriptor block" in the AA$ (or other) variable. The string descriptor block is shown in Figure 21-1.

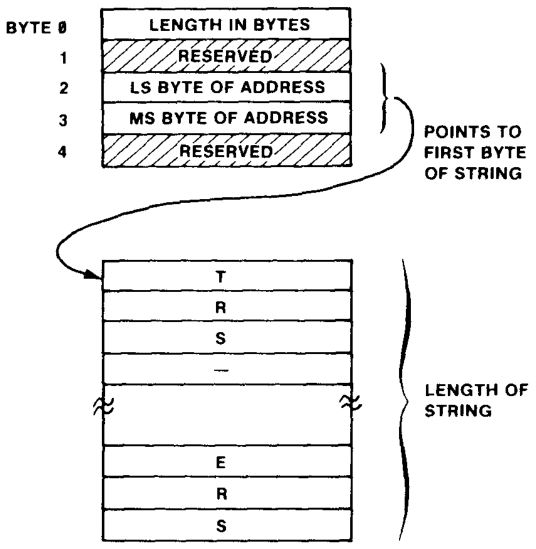


Figure 21-1. String Descriptor Block

The first byte of the string descriptor block is the length of the string in bytes. Each character in the string occupies one byte and the total number of string bytes may be 0 to 255.

The third and fourth bytes of the string descriptor block define the actual address of the string.

Where are strings located?

If you have a string in a BASIC statement, such as

```
100 A$="THIS IS A STATEMENT STRING"
```

then the string will be in the BASIC program statements itself. BASIC statements start in low RAM before variable and other storage and continue to build upward.

If you have a string that is "processed," such as

```
100 A$="THIS IS A PRO"
110 B$="CESSED STRING"
120 C$=A$+B$
```

then the string will be found in the string storage area. This is a temporary storage area for strings that are not present in BASIC program lines. The string storage area is in high memory, just below the protected area for assembly-language subroutines and a BASIC stack.

To pass the location of a string to an assembly-language subroutine, you'd have to do something like:

```
100 SL=VARPTR(ZX$)        'FIND STRING DESC
                           BLOCK LOCATION

120 A=USR(SL)       or     A=USR0(SL)
```

Of course, we've left off all of the other logic concerned with moving the machine-language values and defining the location here. In this case, BASIC would have the location of the **string descriptor block** (not the string) ready to be picked up by a BSR $B3ED.

Using VARPTR With Arrays

When VARPTR is used to find the location of an array, it has the same basic format as with strings. To find the location of array ZX, a numeric array, you would use:

```
100 A=VARPTR(ZX(0))
```

This finds the location of the first element in the array ZX. A numeric array is made up of 5-byte elements.

For a one-dimensional array, the elements start with 0, 1, 2, etc. The 10th element of numeric array ZX, for example, would be 50 bytes after VARPTR (ZX(0)).

For multiple-dimensioned arrays the format is more complicated, and we'll leave it up to you to research (information on array formats is in the BASIC language manual for your Color Computer).

String arrays are not "contiguous" as are the other types of arrays String array descriptor blocks are grouped together in one mass, however, as shown in Figure 21-2.

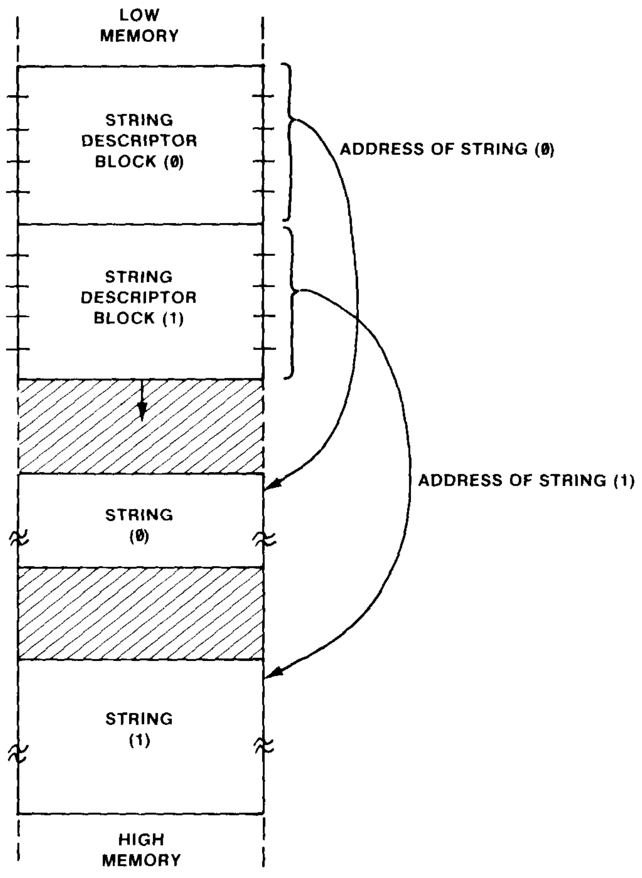


Figure 21-2. String Arrays

To find the location of any string array, just use the index of the array variable as in

```
100 A=VARPTR(A$(5))
```

The address of the string descriptor block will be returned in A.

To show you how VARPTR works, enter the following program, or use the Lesson file:

```
3F00                00100         ORG     $3F00
                    00110 ***********************************************
                    00120 * SUBROUTINE TO PRINT STRING IN REVERSE        *
                    00130 *      ENTRY: STRING DESCRIPTOR BLOCK LOCATION *
                    00140 *      EXIT:  PRINT IN REVERSE ON SCREEN       *
                    00150 ***********************************************
3F00 BD    B3ED     00160 PRTSTR  JSR     $B3ED     GET PARAMETER
3F03 1F    02       00170         TFR     D,Y       MOVE TO Y
3F05 E6    A4       00180         LDB     ,Y        LENGTH OF STRING
3F07 AE    22       00190         LDX     2,Y       GET STRING LOCATION
3F09 3A             00200         ABX               FIND END OF STRING+1
3F0A 108E 050A      00210         LDY     #$500+10  POINT TO SCREEN CENTER
3F0E 5D             00220         TSTB              TEST COUNT
3F0F 27    07       00230         BEQ     PRT090    GO IF "NULL" STRING
3F11 A6    82       00240 PRT010  LDA     ,-X          GET STRING CHARACTER
3F13 A7    A0       00250         STA     ,Y+          STORE ON SCREEN
3F15 5A             00260         DECB                 DECREMENT COUNT
3F16 26    F9       00270         BNE     PRT010       GO IF NOT DONE
3F18 39             00280 PRT090  RTS               RETURN TO BASIC
           0000     00290         END
00000 TOTAL ERRORS

PRT010  3F11
PRT090  3F18
PRTSTR  3F00
```

Figure 21-3. Print String Program

PRTSTR first calls the BASIC subroutine at $B3ED to get the string descriptor block location. It assumes something like A=VARPTR(A$) has been done in the BASIC program, and that the string location is waiting to be picked up.

The string descriptor block location in D is then transferred to Y. B is loaded with the string length, which may be 0.

The next load loads X with the actual address of the string from the 3rd and 4th bytes of the string descriptor block.

X now contains the address of the string, whether it is in a BASIC statement or string variable storage.

The length in B is now added to X by the ABX instruction. The result points to one more than the last character in the string.

Y is loaded with the location of the approximate center character position of video memory. As you recall, the video memory goes from $400 through $5FF, and the center minus a few character positions is at about $500+10.

The string length in B is now tested for 0. If the length is 0, nothing will be printed on the screen, and a jump is made to the return point.

If the string length is not 0, the PRT010 loop prints the string in reverse on the screen, using the string pointer in X, the screen pointer in Y, and the counter in B.

Hints and Kinks 21-1
Notes on PRTSTR

We used the auto decrement in PRTSTR. For that reason, we had X point to the end of the string+1, as we knew that the first LDA ,-X would decrement **first** before the load. Note that we used a TSTB to set the Condition Codes. We mentioned the TSTB early on, but haven't used it for a while. It is simply a one-byte test of A, B, or memory location that sets the Condition Codes without having to do a CMP #0 or other such instruction.

A BASIC program using this code is shown below:

```
100 REM PRINT REVERSE STRING
101 REM EXTENDED COLOR OR DISK BASIC
110 CLS: CLEAR 1000,&H3EFF
120 DATA 189,179,237,31,2,230,164,174,34,58
130 DATA 16,142,5,10,93,39,7,166,130,167,160
140 DATA 90,38,249,57
150 FOR I=16128 TO 16128+24
160 READ A
170 POKE I,A
180 NEXT I
193 DEFUSR0=&H3F00
200 SB=0
210 INPUT A$
220 CLS
230 SB=VARPTR(A$)
243 SR=USR0(SB)
250 GOTO 210
```

Figure 21-4. Print String in BASIC Program

Even if your BASIC code does not look quite the same, the DATA values should be identical.

Run the program above or yours, and you should see any string that is input displayed across the screen in reverse.

An important point about using BASIC programs: **VARPTR locations tend to move!** To use VARPTR properly, you must use it just before the USR call and not introduce "new" variables before the CALL. That's why we defined variable SB in line 200 instead of defining it with the VARPTR call. Any new variable may move all variables down, invalidating a VARPTR location.

Passing Multiple Arguments

We've seen how we can pass a single 16-bit argument or parameter to an assembly-language subroutine and how to pass one argument back. How can we pass several arguments? After all, even the subroutines we used earlier required more than one argument.

There are a number of ways to do this. If the arguments are small enough, then you can pack two into one 16-bit integer number. If we had wanted to clear the video display from a given line and character position, then we might have had something like this:

```
*************************************************
*CLEAR SCREEN FROM LINE X, CP Y                 *
*     ENTRY: (A)=LINE #, 0 — 15                 *
*            (B)=CP, 0 — 31                     *
*************************************************
```

In this case the line values and character position values are small enough to fit into a byte each, and there's no reason why we can't pack them together.

Another way to pass multiple arguments is to use the 16-bit value passed from BASIC as a pointer to a "parameter block." The parameter block could be filled with POKEs from BASIC and could be in a predefined protected area of RAM.

Suppose that we had the following parameters to pass to an assembly-language subroutine that searched a string array for a given string. We might have something like this

Location $3F00=MS byte of first descriptor block

$3F01=LS byte of first descriptor block

$3F02=MS byte of # elements in string array

$3F03=LS byte of # elements in string array

$3F04=MS byte of search string block address

$3F05=LS byte of search string block address

We used an area of protected RAM to hold the parameter block. The first two bytes hold the address of the first descriptor block of the string array. The next two bytes hold the number of elements in the array. The last two bytes hold the address of the search string descriptor block. These values could all have been put into the parameter block by BASIC POKEs.

The search subroutine can output arguments in the same way. It can store the output parameters as follows:

Location $3F06=MS byte of found string or -1

$3F07=LS byte of found string or -1

$3F08=element number if found

The BASIC program can then pick up the output parameters by PEEKs.

Hints and Kinks 21-2
Another Parameter-Passing Option

Another way we could pass parameters is by making the "parameter block" location variable and passing the location to the assembly-language subroutine by the USR call. This would make the subroutine a more "generic" type of subroutine that would process a parameter block in any location. As a rule, it's always wise to make assembly-language code as "general-case" as possible. This type of code is called "parameterized."

Review

To review what we've learned here:

- VARPTR is used in Extended Color or Disk BASIC to find the location of a BASIC variable
- The location of any variable type may be found by VARPTR
- Each string in BASIC is defined by a "string descriptor block" that holds the string length in the first byte and the string location in the third and fourth bytes
- VARPTR locations may change between the VARPTR use and the USR call if previously undefined variables are used
- Multiple arguments may be "packed" into 16 bits if their number ranges are small enough
- Multiple arguments may also be passed via a parameter block in a protected area of memory; this block is used by both BASIC and the assembly-language program as a "common" area
- POKEs and PEEKs can be used by BASIC to store and read values from the parameter block

For Further Study

BASIC PEEK (BASIC manual)

BASIC VARPTR (BASIC manual)

BASIC array formats (BASIC manual)

KEY CHART — CHAPTER 22

INSTRUCTIONS

	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~A~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~B~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~A~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~B~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~D~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~A~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~B~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~CC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~A~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~B~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~A~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~B~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~C~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~S~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~L~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~Q~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~E~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~T~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~ERENT~~
~~ECT~~
~~ENDED~~
~~EDIATE~~
~~PLE INDEXED~~
~~ATIVE~~
~~LACEMENT INDEXED~~
~~O INCREMENT/DECREMENT~~
RECT
HISTICATED

PSEUDO OPS

~~ORG~~
~~RMB~~
SET
~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERRORS~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	→ BRANCH INDIRECT
~~N(UMERIC) MODE~~	; ~~FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	/ ~~EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	; ~~SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	~~PASSING PARAMETERS~~
~~INCREMENTS/DECREMENTS~~	~~VARPTR USE~~
~~COMPLEMENTS~~	**ROM SUBROUTINES**
~~LOGICAL OPERATIONS~~	OTHER ADDRESSING
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Type Present Chapter
lar Type = Future Chapters
~~Type~~ = *Past Chapters*

Chapter 22
Using ROM Subroutines

ROM subroutines are subroutines in the BASIC interpreter program that are available for the assembly-language user. The ROM subroutines perform useful functions such as reading a keyboard character, displaying a character on the screen, or reading a cassette byte. The BASIC interpreter has many different subroutines that might be usable, but the ones we'll be discussing here are subroutines that are "documented" in Radio Shack documentation. It would be very difficult to document **all** possible subroutines. Later revisions to the BASIC interpreter might be very difficult if all subroutines had to remain fixed in both location and input and output parameters.

There is a list of many ROM subroutines in the user manuals for EDTASM+, Extended Color BASIC, and other Radio Shack manuals. We'll just be working with two ROM subroutines here. They are

1. Get Keyboard Character, (address at) Location $A000 (indirect)
2. Display a Character, (address at) Location $A002 (indirect)

Cautions On Using ROM Subroutines

One of the most important points that we can make about ROM subroutine use is that you must be aware of which registers the subroutine uses. This is true for **any** subroutine called, ROM subroutine or not, although we didn't stress this too greatly in previous chapters.

The Get Keyboard Character subroutine, for example, returns the code of the next key pressed on the keyboard in the A register. In doing so, it alters the U and S registers.

The Display Character subroutine displays the ASCII character in A, but in doing so, it also alters the Condition Codes.

If you are using registers to hold a pointer value, or any other quantity, it may be destroyed after a return is made from typical ROM subroutines. Prior to calling the subroutines, therefore, you must PSHS or PSHU any registers (or Condition Codes) that you want to preserve.

As a matter of fact, if you are unsure of which registers are used in a subroutine, there's no reason why you can't PSH all of the registers, or at least all of the ones you're using. To save all registers takes only 1 instruction:

```
        PSHS    CC,A,B,DP,X,Y,U,PC SAVE REGISTERS
```

Using Display a Character and Get Keyboard Character

Display a Character

This ROM subroutine is entered with a display character in the A register and with a 0 in RAM location $6F (DEVNO). The DEVNO location is a "working storage" location used by BASIC to define to what device the character should be sent; 0 is the screen and -2 is the printer. The subroutine displays the character at the current screen location and then returns to the user program. Doesn't sound like much, does it? In fact, though, you have all the power of the Display Driver at your disposal through this entry point.

Using Display a Character is as simple as it looks. Put a 0 in $6F, put the character in A and JSR "indirect" to location $A002. Use SHIFT, DOWN ARROW for left bracket and SHIFT, RIGHT ARROW for right bracket to denote indirect addressing.

Get Keyboard Character
This ROM subroutine is entered with no parameters and returns to the user program with an ASCII character representing the next keypress in the A register, or with a zero if no key was pressed. The character entered is not displayed on the screen. A JSR indirect to location $A000 is done for this ROM subroutine (JSR [$A000]).

Here again, this doesn't seem like a very powerful subroutine, but don't forget that implicit in this call are dozens of bytes of instructions, including keyboard debouncing, "n" key rollover, and other processing.

With only these two ROM subroutines as a base, you can build a whole series of your own subroutines that can translate character strings, perform special display functions, do word processing, and other applications.

A Simple Text Editor

To show you how these two subroutines can be used to build on, we've written a simple text editor. This program will utilize the Get Keyboard Character and Display Character subroutines to implement a "stand-alone" text editor that will allow you to enter text and store it on the screen. The screen cursor is controlled by the up arrow, down arrow, right arrow, and left arrow keys; you may move the cursor anywhere on the screen that you wish to initiate new text, or to overwrite old.

```
3F00                00100         ORG     $3F00
                    00110 *********************************************
                    00120 * MINI TEXT EDITOR                          *
                    00130 *********************************************
3F00 BD   A928      00140 MINITE  JSR     $A928   CLEAR SCREEN
3F03 0F   6F        00150         CLR     $6F     INITIALIZE DISPLAY
3F05 AD   9F A000   00160 TXT010  JSR     [$A000]   INPUT CHARACTER
3F09 27   FA        00170         BEQ     TXT010    GO IF 0
3F0B 8E   3F2A      00180         LDX     #FTAB   FUNCTION KEY TABLE
3F0E C6   04        00190         LDB     #4      SIZE OF TABLE
3F10 A1   80        00200 TXT012  CMPA    ,X+       SEARCH FOR KEY
3F12 27   09        00210         BEQ     TXT020    GO IF FOUND
3F14 5A             00220         DECB              DECREMENT COUNT
3F15 26   F9        00230         BNE     TXT012    GO IF NOT END
3F17 AD   9F A002   00240 TXT015  JSR     [$A002] NOT FOUND, TEXT
3F1B 20   E8        00250         BRA     TXT010  GO FOR NEXT CHARACTER
3F1D 1F   10        00260 TXT020  TFR     X,D     PNTR TO D
3F1F 83   3F2B      00270         SUBD    #FTAB+1 FIND INDEX
3F22 58             00280         LSLB            INDEX*2
3F23 C3   3F2E      00290         ADDD    #BTAB     POINT TO SUBROUTINE
3F26 1F   01        00300         TFR     D,X       NOW IN X
3F28 6E   94        00310         JMP     [,X]      JMP OUT
3F2A      5E        00320 FTAB    FCB     $5E     UP ARROW
3F2B      0A        00330 FTABP1  FCB     $0A     DOWN ARROW
3F2C      09        00340         FCB     $09     RIGHT ARROW
3F2D      08        00350         FCB     $08     LEFT ARROW
3F2E      3F36      00360 BTAB    FDB     UPARR   UP ARROW PROCESS
3F30      3F3B      00370         FDB     DWNARR  DOWN ARROW PROCESS
3F32      3F40      00380         FDB     RGTARR  RIGHT ARROW PROCESS
3F34      3F45      00390         FDB     LFTARR  LEFT ARROW PROCESS
3F36 CC   FFE0      00400 UPARR   LDD     #-32    ONE LINE BACK
3F39 20   0D        00410         BRA     CURSOR  GO TO CHANGE CURSOR
3F3B CC   0020      00420 DWNARR  LDD     #+32    ONE LINE FWARD
3F3E 20   08        00430         BRA     CURSOR  GO TO CHANGE CURSOR
3F40 CC   0001      00440 RGTARR  LDD     #+1     ONE CP FORWARD
3F43 20   03        00450         BRA     CURSOR  GO TO CHANGE CURSOR
3F45 CC   FFFF      00460 LFTARR  LDD     #-1     ONE CP BACK
3F48 D3   88        00470 CURSOR  ADDD    $88     CHANGE CURSOR LOC
3F4A DD   88        00480         STD     $88     STORE (MAY BE CHANGED)
3F4C 1083 0400      00490         CMPD    #$400   TEST FOR BEFORE START
3F50 24   07        00500         BHS     CUR020  GO IF NOT BEFORE
3F52 C3   0200      00510         ADDD    #$200   WRAP AROUND
3F55 DD   88        00520 CUR010  STD     $88     STORE
3F57 20   AC        00530         BRA     TXT010  GO FOR NEXT CHARACTER
3F59 1083 05FF      00540 CUR020  CMPD    #$5FF   TEST FOR BEYOND END
3F5D 23   A6        00550         BLS     TXT010  GO IF OK
3F5F 83   0200      00560         SUBD    #$200   WRAP AROUND
3F62 20   F1        00570         BRA     CUR010  GO TO STORE
          0000      00580         END
00000 TOTAL ERRORS

BTAB     3F2E
CUR010   3F55
CUR020   3F59
CURSOR   3F48
DWNARR   3F3B
FTAB     3F2A
FTABP1   3F2B
LFTARR   3F45
MINITE   3F00
RGTARR   3F40
TXT010   3F05
TXT012   3F10
TXT015   3F17
TXT020   3F1D
UPARR    3F36
```

Figure 22-1. Text Editor Program

If you want to run this program, assemble it by normal means and execute from MINITE.

You'll see the program clear the screen and position the cursor in the upper left-hand corner of the display, the HOME position. You can now enter text and move the cursor around with the arrows.

The Get Keyboard Character subroutine at [$A000] returns a code corresponding to the key pressed. Usually this is an ASCII code corresponding to an alphabetic character, numeric character, or special character, such as "#."

The subroutine, however, also returns codes for special keys, such as the arrow keys, ENTER, and others. You'll find a complete list in the back of your BASIC manual. The ones we'll be considering here are the codes for up arrow ($5E), down arrow ($0A), right arrow ($09), and left arrow ($08).

These keys will initiate special actions in MINITE to move the screen cursor. The cursor position is held in a BASIC RAM variable at location $88 (two bytes). Since we'll be using the display driver of BASIC in MINITE we'll have to "maintain" this variable to control the cursor position.

The $88 variable holds the current cursor position in terms of the absolute address of the text screen from $400 through $5FF. We can change this variable to change the cursor position.

Let's take a more detailed look at the program:

TXT010 starts the main loop of the program.

A ROM subroutine at $A928 (not indirect) is called to initialize the display. This subroutine also establishes the current cursor position as upper left ($400 at $88).

A zero is stored at location $6F to initialize the device type to the display.

The next keypress is input from the Get Keyboard Character subroutine. A scan is then made through the FTAB table to see if the input character matches any code in the table. There are 4 codes in the table, corresponding to the arrow key codes. At the end of the scan, the X register points to the character if the character has been found (BEQ).

If the character has not been found, then the character is a "normal" text character. It's output to the display by CALLing $A002 "indirect" at TXT015. Note that up to this point, no character has been output; reading in the character does not automatically display it. After the output, a loop is made back to TXT010 to input and display the next character.

If the character is found in the FTAB, then the start address of FTAB is subtracted from the contents of D, into which the X register has been transferred. D now contains an index of 0, 1, 2, or 3. This index value is doubled by a Logical Shift Left of B and added to the starting address of BTAB, a "branch table." At the end of the add, D points to a branch address corresponding to the processing for the special key. D is now transferred back to X. A JMP [,X] "indirect jump" then causes a jump not to the BTAB, but indirectly to the address specified in the BTAB location.

Each of the 4 cursor control processing routines adjusts location $88 to the next or previous line or next or previous character position by adding or subtracting a "displacement" for the position in the text screen. If the position is before the beginning of the screen or after the end of the screen, $200 is added or subtracted to cause a "wrap around" of the cursor position.

Hints and Kinks 22-1
The Clear Screen ROM Subroutine

The Clear Screen ROM subroutine is one of those "undocumented" ROM subroutines we've been telling you about. We're taking a chance in using it here, but we've thrown caution to the winds. Location $A928 should hold a $C6, and the next location should be a $60, as a double check on the ROM version. If you don't see these values, "NOP" the ROM call, or leave it out on assembly.

Hints and Kinks 22-2
Screen Wraparound

The effect of adding $200 if the screen cursor position is before $400 is to move the cursor to the end of the screen, but at the same position as the "phantom line" before the first screen line.

The effect of subtracting $200 if the screen cursor position is after $5FF is to move the cursor to the beginning of the screen, but at the same position as the "phantom line" after the last screen line.

This effect is known as "screen wraparound."

Hints and Kinks 22-3
Indirect Addressing

You can see how indirect addressing works from the JMP [,X] example. The jump was made to the "effective address," the address defined by the contents of the location determined by ,X. Anytime you see an address enclosed by left and right brackets, it will mean indirect addressing is in force. (The left bracket is a SHIFT, DOWN ARROW and the right bracket is a SHIFT, RIGHT ARROW.) We'll discuss this mode more in the next chapter.

Using ROM Subroutines for Your Own Code

The simple program above shows you how you can take advantage of some of the existing ROM subroutines to eliminate a lot of tedious coding. Look for other examples of keyboard input processing, display output, line printer output, cassette operations, and disk operations in your BASIC and other manuals.

Review

To review what we've learned in this chapter:

- There are a number of documented ROM subroutines that can be used to eliminate your own assembly-language coding for keyboard input, display operations, and others
- When using these subroutines, or any subroutines, you must be aware of which registers may be destroyed by the action of the subroutine; save these registers by PSHS or PSHU before the subroutine call
- Display Character subroutine outputs one character to the video display, and uses the full logic of the BASIC display driver software
- Get Keyboard Character inputs the next keypress from the keyboard input driver or a 0 if no key has been pressed

For Further Study

Character codes for input and output (BASIC manual)
ROM subroutines (BASIC and other manuals)

KEY CHART — CHAPTER 23

INSTRUCTIONS

~~X~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~CA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~CB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~DA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~DB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~DD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~DA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~DB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~DCC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~LA~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~LB~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
~~L~~	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~RA~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~RB~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
~~R~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~C~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~CC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~S~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~CS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~Q~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~EQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~E~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~GE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~T~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~GT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~I~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~HI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~S~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~HERENT~~
~~RECT~~
~~TENDED~~
~~MEDIATE~~
~~MPLE INDEXED~~
~~LATIVE~~
~~SPLACEMENT INDEXED~~
~~TO INCREMENT/DECREMENT~~
DIRECT
PHISTICATED

PSEUDO OPS

U	~~ORG~~
~~B~~	~~RMB~~
~~C~~	SET
~~B~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERRORS~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	→ **BRANCH INDIRECT**
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	~~' FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	~~PASSING PARAMETERS~~
~~INCREMENTS/DECREMENTS~~	~~VARPTR USE~~
~~COMPLEMENTS~~	~~ROM SUBROUTINES~~
~~LOGICAL OPERATIONS~~	**OTHER ADDRESSING**
~~MULTIPLE PRECISION~~	GRAPHICS
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

d Type **Present Chapter**
ular Type Future Chapters
~~c Type~~ · *Past Chapters*

Chapter 23
Addressing Modes, EQU and SET

In this chapter we're going to fill in the addressing modes that we mentioned briefly or not at all. The addressing modes that we've been using up to this points are the most common — inherent, immediate, extended, direct, relative, and indexed — but there are a lot of variations of these modes and several modes that we haven't covered that are very powerful.

We'll also talk about about two EDTASM+ pseudo ops that we've neglected, EQU and SET.

Using the S and U Stack Pointers with Indexing

Most of the instructions using indexing used either the X or Y registers as index registers. However, in the general case, X, Y, S, or U can be used as index registers interchangeably. If you'll look at Appendix II under "Indexed Addressing Modes", you'll see that the "R" opposite the indexing type can stand for any of the four registers.

S and U With Offsets

S and U point to the last used stack location, as shown in Figure 23-1. When a signed displacement type indexed operation is done with S, data in the stack can be accessed very easily. By "data in the stack" we mean data that has been pushed into the stack that has not yet been retrieved. All data from the stack pointer (S) address **up to the original top of stack** is valid. All data from the stack pointer address -1 and below is invalid. Why?

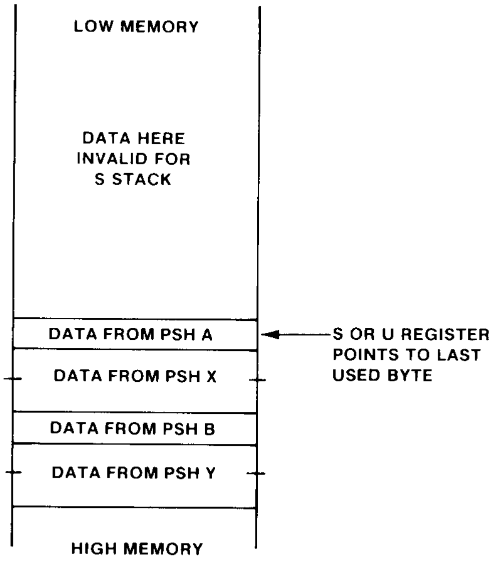


Figure 23-1. S and U Register Use

The reason that data "below" the stack is invalid is that the S stack is constantly being used, not only by code in your own program, but possibly by interrupt processing routines that use the S stack for return addresses and condition codes. As good programming practice, **never** assume that data "out of the stack" (below the current stack pointer location) is valid.

Hints and Kinks 23-1
Stack Philosophy

The whole purpose of the stack is to create an area in memory that can be used for storage of data and addresses by any number of subroutines or interrupt routines without conflict. The stack pointer always points to the next available byte to be used and never destroys any data for a "higher-level" subroutine or interrupted code segment. This permits such techniques as "recursiveness," where a code segment calls itself as a subroutine, and "reentrancy" where multiple interrupts or tasks can use the same code segment; in both cases stack storage of results will be non-conflicting.

Generally, then, even though the S register may be used as an index register, it is normally used to retrieve data that exists in the stack and not beyond. This amounts to using S as an index register only with a positive displacement to address data "in the stack" (see Figure 23-2).

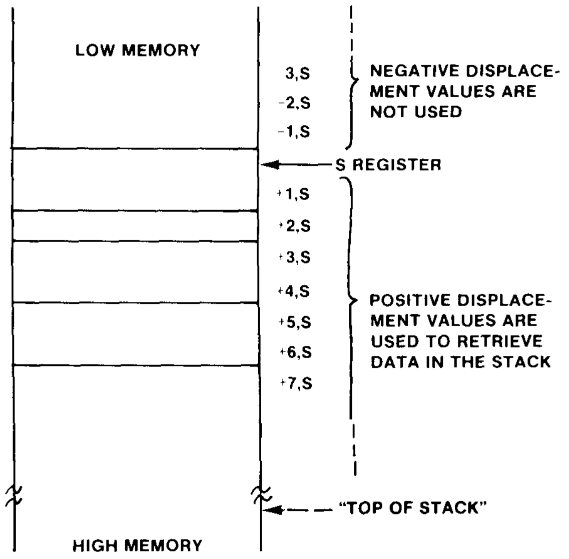

LOW MEMORY
3,S
-2,S
-1,S
NEGATIVE DISPLACEMENT VALUES ARE NOT USED
S REGISTER
+1,S
+2,S
+3,S
+4,S
+5,S
+6,S
+7,S
POSITIVE DISPLACEMENT VALUES ARE USED TO RETRIEVE DATA IN THE STACK
"TOP OF STACK"
HIGH MEMORY

Figure 23-2. Using the S Register as an Index Register

The U stack, on the other hand is not used by "hardware." An interrupt processing program should save the "state" of the user stack, and the U register can be used for indexing both forward and back from the base address. See Figure 23-3.

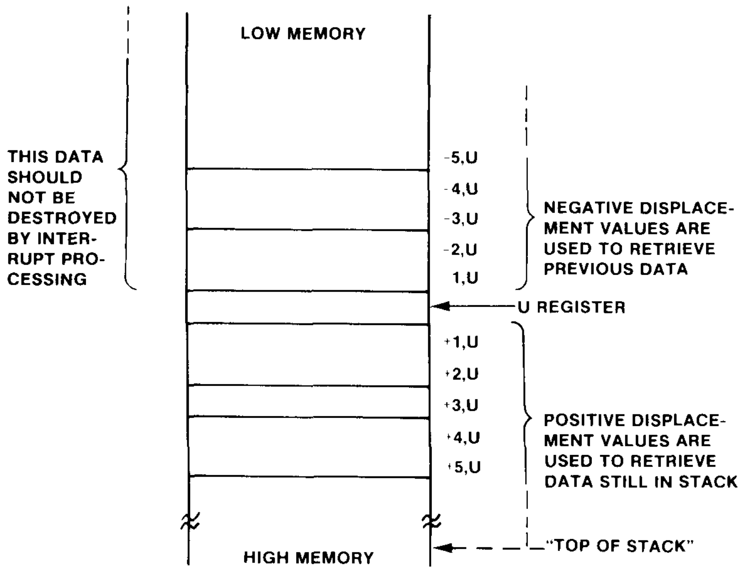

LOW MEMORY
THIS DATA SHOULD NOT BE DESTROYED BY INTERRUPT PROCESSING
-5,U
-4,U
-3,U
-2,U
1,U
NEGATIVE DISPLACEMENT VALUES ARE USED TO RETRIEVE PREVIOUS DATA
U REGISTER
+1,U
+2,U
+3,U
+4,U
+5,U
POSITIVE DISPLACEMENT VALUES ARE USED TO RETRIEVE DATA STILL IN STACK
"TOP OF STACK"
HIGH MEMORY

Figure 23-3 Using the U Register as an Index Register

Suppose that we pushed the A, X, and Y registers into the S stack, and then did a BSR to a subroutine. We could "pass parameters" by accessing the stack data from the subroutine by indexing. An example is shown here:

```
SUBROU  LDA   +2,S        GET A PARAMETER
        LDX   +3,S        GET X PARAMETER
        LDY   +5,S        GET Y PARAMETER
```

The three instructions pick up the data in the same order it was put into the stack. We started out with a +2 to bypass the return address, which is stored at ,S. If we had done a

```
        PULS  A,X,Y       GET THREE PARAMETERS
```

it would have picked up the wrong data, as the first two bytes in the stack are the return address. The "index" operation works out very nicely for picking up the parameters from the stack without "resetting" the stack.

Returning From a Subroutine

The "normal" way to return from a subroutine is to execute an

```
        RTS
```

which pulls the return address from the stack, storing it into the Program Counter, and causing an automatic branch back to the instruction after the BSR, LBSR, or JSR.

An alternative way to return, however, is to "purge" the stack of data pushed into the stack in the subroutine in addition to loading the PC. Here's an example:

```
SUBROU  PSHS   X,Y       SAVE X AND Y
        LBSR   NEWSUB    GO TO STORE SUBROUTINE
        PULS   X,Y,PC    RECOVER X,Y, RETURN
```

This subroutine "saves" the X and Y registers and then calls another subroutine NEWSUB. On return from NEWSUB, the PULS not only restores X and Y, which may have been restored in NEWSUB, but also loads the PC with the return address to effectively do an RTS.

Hints and Kinks 23-2
When Do You Save Registers?

Cpu registers are saved in the "main line" code before calling a subroutine, or in the subroutine itself. We're talking about saving registers in the stack, of course, and saving registers that are used in the subroutine for processing. Since the 6809 has a limited number of registers, compared to such microprocessors as the Z-80, it's probably best to assume that all registers will be destroyed in subroutine processing, and to save any important results in the stack before the subroutine is called.

Using Auto Increment/Decrement

Continuing with the "Indexed Addressing Modes" of Appendix II, let's consider Auto Increment and Decrement. Here again, the "R" register that can be automatically incremented or decremented is any of the 4 registers X, Y, U, or S.

We've used X and Y frequently in this mode. Recall that

```
LDA     ,X+
LDB     ,Y++
```

would increment X by one count after A was loaded with the memory byte pointed to by the X index register, while Y would be incremented twice after B was loaded with the memory byte pointed to by the Y register and that:

```
LDA     ,-X
LDB     ,--Y
```

would **decrement** X by one count and load A with the contents of the effective address, while Y would be decremented twice before B was loaded with the contents of the effective address.

The same technique can be used with U and S. Auto increment of S or U would **reset** the stack by one or two bytes, while auto decrement would create one or two more "unused" bytes in the stack:

```
LDA     ,S+       GET STACK BYTE (PULS  A)
STB     ,U-       SAME AS PSHU  B
```

In fact, the most common use of auto increment/decrement will be X or Y, followed by S, followed by U (least usage).

Accumulator Offset from R

This addressing mode is one that we haven't used previously. Think of it as the same as a "displacement" (offset) indexed mode with the displacement not in the instruction, such as

```
LDA     +100,X
```

but in either the A, B, or D register. If the A register contained $FC and the B register contained +5, then the instructions:

```
LDY     A,X
LDY     B,X
```

would be exactly equivalent to:

```
LDY     -4,X
LDY     +5,X
```

In other words, the value in A or B is used as a displacement value to be added to the contents of the index register to form the effective address.

The D register can also be used in this format. If D contained -1000, the instruction

```
LDY     D,X
```

would be equivalent to

```
LDY     -1000,X
```

These forms of displacement (offset) indexing won't be used as frequently as the type where the displacement is specified in the instruction, but are handy to use where A, B, or D contain an "index" value.

Here again, the index registers that can be used for this type of addressing are X and Y, but also (less frequently) U and S.

Program Counter Relative

This is another addressing mode type that we may have mentioned briefly, but did not cover in detail. In this mode, the **effective address** is formed by adding an 8- or 16-bit (no 5-bit) displacement value from the instruction to the contents of the Program Counter register.

Take this code sequence, for example:

```
        ORG     $3F00
START   LDA     COUNT       GET COUNT
        LDA     COUNT,PCR   GET COUNT
        JMP     NEXT        BYPASS DATA
COUNT   FCB     23          DATA=23
NEXT    JMP     NEXT        LOOP HERE
        END
```

This is a nonsensical code sequence as it loads the A register twice, but it's worth assembling to see what happens:

The assembly is shown in Figure 23-4.

```
3F00                 00100          ORG    $3F00
3F00 B6    3F0A      00110 START    LDA    COUNT      GET COUNT
3F03 A6    8D 0003   00120          LDA    COUNT,PCR GET COUNT
3F07 7E    3F0B      00130          JMP    NEXT       BYPASS DATA
3F0A       17        00140 COUNT    FCB    23         DATA=23
3F0B 7E    3F0B      00150 NEXT     JMP    NEXT       LOOP HERE
           0000      00160          END
00000 TOTAL ERRORS

COUNT    3F0A
NEXT     3F0B
START    3F00
```

Figure 23-4. Program Counter Relative Program 1

The first instruction is our old friend, extended addressing, which assembles with an absolute address in the instruction itself.

The second instruction, though, assembles as $A6, $8D, $00, $03. The first byte of this is the opcode for the LDA, indexed $A6. The second byte is $8D, which tells the 6809 that this is Program Counter Relative, 16-bit displacement. The third and fourth bytes are the most interesting. When these bytes are added to the contents of the PC, the result will be the effective address of the operand. Here we've got $3F07 in the PC. Adding $0003 gives us $3F0A, the address of COUNT.

Now change the ORG to $3E00 and reassemble. The result is shown in Figure 23-5. The instruction is **still** $A6, $8D, $00, $03! Since this instruction is always relative to the contents of PC, it will never change and will always be **relocatable.** The operand is defined relative to the contents of the PC.

```
3E00                 00100          ORG    $3E00
3E00 B6    3E0A      00110 START    LDA    COUNT      GET COUNT
3E03 A6    8D 0003   00120          LDA    COUNT,PCR GET COUNT
3E07 7E    3E0B      00130          JMP    NEXT       BYPASS DATA
3E0A       17        00140 COUNT    FCB    23         DATA=23
3E0B 7E    3E0B      00150 NEXT     JMP    NEXT       LOOP HERE
           0000      00160          END
00000 TOTAL ERRORS

COUNT    3E0A
NEXT     3E0B
START    3E00
```

Figure 23-5. Program Counter Relative Program 2

Note that we didn't have to specify a displacement in this instruction. We only had to specify the label of the operand; the EDTASM+ assembler located the label of the operand and then calculated the proper displacement, putting it into the instruction displacement byte.

When would you use this type of addressing? Anytime that you wanted your code to be "position independent" or "relocatable." It should be used for LDs and STAs, and other instructions which would normally assemble in extended addressing mode. Note that to make an existing instruction of this type fully position independent, all you have to do is to add a ",PCR" at the end:

```
        LDA     OP1         LOAD OP 1
        LDB     OP2         LOAD OP 2
```

becomes:

```
        LDA     OP1,PCR     LOAD OP 1, POS IND
        LDB     OP2,PCR     LOAD OP 2, POS IND
```

Hints and Kinks 23-3 "Relocatable" Vs. "Position Independent"

"Position Independent" is probably a better term to describe code that can be executed in any portion of memory without reassembling. Code can be "relocated" by adding a suitable "bias" to every absolute address, and there might be some confusion over this technique and "position independent" relocatable code.

Indirect Addressing

In indirect addressing, an extended mode address or an indexed mode address is first computed by normal means. This address is then used to retrieve a 16-bit address from memory, which then becomes the effective address for the instruction. We saw an example of this in the last chapter when we had a table of addresses:

```
BTAB    FCB     UPARR
        FDB     DWNARR
        FDB     RGTARR
        FDB     LFTARR
```

We wound up with a pointer to a BTAB entry in the X register. We were then able to JMP out to the processing routine UPARR, DWNARR, RGTARR, or LFTARR not by jumping to the BTAB address, which would have resulted in invalid instructions, but by jumping "indirectly" to the address of the table. The JMP instruction was a

```
                JMP     [(,X)]
```

where left and right brackets were used to define the "indirectness."

Indirection can be used in extended addressing such as:

```
                LDA     [STDATA]      GET DATA VALUE
                JMP     NEXT          GO TO NEXT
        STDATA  FDB     TABLE+1       POINT TO TABLE
                .
                .
                .
        TABLE   FCB     17
                FCB     8
```

which loads A not with the constant "TABLE+1" in STDATA, but with the value found in TABLE+1, a value of 8.

Indirection can also be used in most indexing modes, such as auto increment/decrement and Program Counter Relative. A complete list is shown in Appendix II under "Indexed Addressing Modes."

Indirection is always indicated by left and right brackets, generated by a SHIFT, DOWN ARROW and a SHIFT, RIGHT ARROW, respectively. Note that the brackets are different from left and right parentheses.

When Is Indirect Addressing Used?

In simple programs you probably won't be using indirect addressing much. It is handy for indirect JMPs and for defining **pointers** "dynamically." We've tried to present the addressing modes in a manner that reflected the amount of use in less complicated programs.

Even in more complicated programs, indirect addressing will probably be rarely used, but it can save three or four instructions spent loading an address value into an index register and will not waste a cpu register in the process.

The EQU and SET Pseudo-Ops

We've covered all pseudo ops in EDTASM+ with the exception of two, EQU and SET. These pseudo ops are used to set a label equal to some value. EQU is a one time definition, while SET can be used several times.

Suppose that you wanted to use the carriage return (ENTER) character repeatedly to cause a new line to be displayed or printed. You could say:

```
    CR          EQU     13              DEFINE CR
```

Thereafter, any time you used the symbol "CR" the code 13 would be used in assembly, as in

```
                LDA     #CR             LOAD 13 INTO A
```

The EQU is kind of a "forced symbol table entry" that puts the EQU label into the symbol table with the value assigned to it in the operand field.

Here's another example of an EQU:

```
        LDB     #TABLSZ     LOAD TABLE SIZE
        .
        .
        .
TABLE   FCB     12          TABLE OF CONSTANTS
        FCB     13
        FCB     8
        FCB     10
        FCB     1
TABLSZ  EQU     *-TABLE     COMPUTE TABLE SIZE
```

In this case an EQU was used to set label "TABLSZ" equal to the size of the TABLE. The "*" character is a special EDTASM+ character that means "the value of the current assembly location." If TABLE were at assembly location $3F00, for example, "*" would be at location $3F05 when the EQU was encountered, and TABLSZ would be set equal to $3F05-$3F00, or 5.

EQUates can also be used with addresses, as in:

```
TABLE   FCB     12          TABLE OF CONSTANTS
        FCB     13
        FCB     8
        FCB     10
        FCB     1
TABLSZ  EQU     *-TABLE     COMPUTE TABLE SIZE
TABLE1  EQU     TABLE       TABLE1=TABLE
```

which would make TABLE and TABLE1 synonymous.

Hints and Kinks 23-4
More on the Asterisk

The asterisk, as we've mentioned, stands for the current assembler location. A common technique for equating one location with several labels, or defining fields in a table goes like this:

```
TABLE   EQU     *
EMPNO   EQU     *
        RMB     3
EMPNAM  EQU     *
        RMB     25
```

What we've done here is to define a table as "TABLE" at the current assembler location, say $3F00. "EMPNO" would also be defined as

$3F00. "EMPNAM" would be defined 3 bytes later at $3F03. The entire table would contain 28 bytes.

The SET pseudo op performs the same function as the EQU, except that SET can be used more than once. If the CR character were to be redefined for a different type of printer, for example, you might have

```
CR          SET        13       DEFINE CR FOR PRINTER 1
             .
             .
CR          SET        10       DEFINE CR FOR PRINTER 2
```

SET is seldom used compared to EQUates.

Hints and Kinks 23-5
Branch Indirect

There's a ZBUG "one-time" typeout that allows you to "open" a location "indirectly." Suppose that you are examining in ZBUG and have the following sequence:

```
3002/ CMPA #23
3004/ BNE 300F
3006/ JSR >35A5
```

You'd like to examine the sequence at 35A5 at this point to see what instructions are there. You can easily do this by using the RIGHT ARROW "branch indirect" command, which will "open" a new location in a kind of indirect examine:

```
3006/ JSR >35A5                (RIGHT ARROW)
35A5/ CMPA #17
35A7/ BRA 3600
```

Review

To recap what we've learned in this chapter:

- The S and U registers can be used fully as index registers in the same fashion as X and Y
- When S is used as an index register, take care not to go "out of stack"
- The S and U registers can also employ auto increment/decrement
- Program Counter Relative addressing creates instructions that are fully relocatable and are relative to the current PC contents

- Indirect addressing obtains the contents of a memory address computed within the brackets and uses it as an effective address for the actual operand
- The EQU pseudo op sets a label equal to an operand, which may be another label, numeric value, or expression

KEY CHART — CHAPTER 24

INSTRUCTIONS

~~ABX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~ADCA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~ADCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~ADDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~ADDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~ANDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~ANDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ANDCC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~ASLA~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~ASLB~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
~~ASL~~	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~ASRA~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~ASRB~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
~~ASR~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~BCC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BCS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~LBEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~BGE~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~LBGE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~BGT~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~LBGT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TAB~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERROR~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOC~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDIN~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	~~→ BRANCH INDIRECT~~
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BY~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
~~DISPLACEMENT INDEXED~~
~~AUTO INCREMENT/DECREMENT~~
~~INDIRECT~~
~~SOPHISTICATED~~

PSEUDO OPS

~~EQU~~	~~ORG~~
~~FCB~~	~~RMB~~
~~FCC~~	~~SET~~
~~FDB~~	~~SETDP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	~~PASSING PARAMETERS~~
~~INCREMENTS/DECREMENTS~~	~~VARPTR USE~~
~~COMPLEMENTS~~	~~ROM SUBROUTINES~~
~~LOGICAL OPERATIONS~~	~~OTHER ADDRESSING~~
~~MULTIPLE PRECISION~~	**GRAPHICS**
~~DATA VALUES~~	SOUND
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Bold Type Present Chapter
Regular Type - Future Chapters
~~*Italic Type*~~ - *Past Chapters*

Chapter 24
Assembly-Language Graphics

In this chapter we'll take a brief look at assembly-language graphics on the Color Computer. We'll review the way memory is laid out for graphics, look at some of the graphics modes, and finally see how we can implement a "Times 4 Zoom" feature for graphics mode 4.

BASIC Vs. Assembly Language

In the following discussion we're going to assume that you'll be interfacing to Extended Color BASIC with short assembly-language routines. There's a good reason for this — the Extended Color BASIC commands for graphics — LINE, "box," "filled in box," CIRCLE, "ellipse," "arc," DRAW, PAINT, and the rest are so powerful that to duplicate them in assembly language would be very difficult. After all, these functions are in assembly language, and operate fairly efficiently. Therefore, we'll assume that you'll be using Extended Color BASIC to create graphics with some help from short assembly-language code to create new functions that aren't found in BASIC.

Memory Layout for Graphics

First of all, let me put a plug in for my Radio Shack book "Color Computer Graphics." If you are going to be doing anything in graphics, it's a quick reference guide to using the Color BASIC and Extended Color BASIC graphics commands, graphics architecture, and graphics routines. We won't cover graphics in detail here, as much of the material is already in "Color Computer Graphics."

There are, as you probably know, two different display areas, one for "text" and one for "graphics." These are shown in Figure 24-1.

Note

The graphics pages start at location $E00 in Disk BASIC systems. This chapter is referenced to a non-disk system.

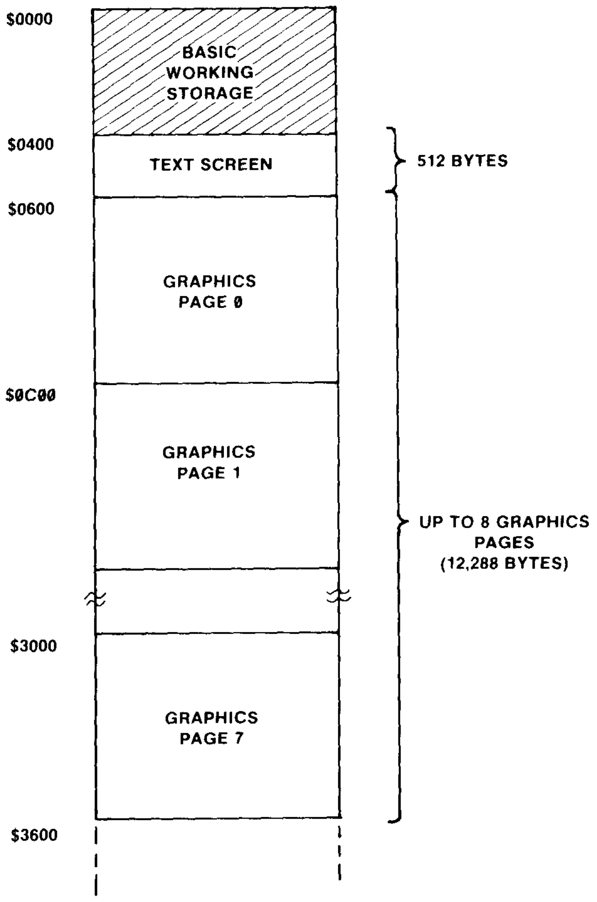


Figure 24-1 Display Areas

The first area is the text screen, located at $400 and going up to $5FF, a total of 512 bytes representing the 16 rows of 64 characters each.

The second area contains the graphics pages of 1536 bytes each and starts at $600. Depending upon how many pages you allocate by the Extended Color BASIC PCLEAR command (1 to 8), there may be from 1536 to 12,288 bytes of graphics area. The "default" amount is 4 graphics pages, or 6,144 bytes.

Any area above the last graphics page is used for storage of BASIC text, variables, etc.

The Text Screen

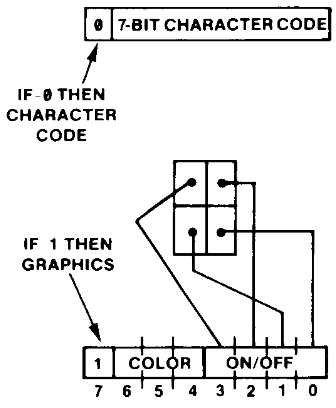
The text screen is divided into 512 "character positions," 16 lines of 32 characters each. Each of these character positions may represent either an alphanumeric or special character or a limited graphics character. We say "limited" because the graphics in the **implemented** modes for the text screen are just not as dense as the graphics in the graphics pages.

There is one byte in memory for every character position on the text screen for the one "implemented" "semi-"graphics mode. (There are 4 unimplemented "semi-"graphics modes for the text or graphics screens that allow high resolution, but at the expense of complicated "mapping" of more than one byte per character position; see my book.)

These bytes follow a one-for-one mapping with the character position on the screen. The byte at $400 is line 0, cp 0, the byte at $401 is line 0, cp 1, etc.

If the high-order bit of the memory byte is a 0, then the byte represents an alphanumeric or special character. These characters are internally generated by a "VDG" or Video Display Generator chip which contains the dot matrix layout of each character and controls the graphics logic of the Color Computer.

If the high-order bit of the memory byte is a 1, then the byte represents graphics data as shown in Figure 24-2. The first bit is a 1, the next three bits are the color code 000 through 111, and the last four bits are the element on/off status.



000	GREEN
001	YELLOW
010	BLUE
011	RED
100	BUFF
101	CYAN
110	MAGENTA
111	ORANGE

Figure 24-2. Text Screen Character Coding

Because there is only one color code for each byte, you can see that all of the 4 elements per character position must be the same color, even though there are 2048 separate elements per text screen when all character positions have been set to graphics mode.

You can control the text screen very easily in assembly language by simply storing the proper bytes in text screen memory. To draw a thin line at the top of the screen, for example, you'd do:

```
        LDX   #$400       POINT TO SCREEN START
        LDA   #$BC        RED, TURN ON ELEMENTS
DRAWL   STA   ,X+             DRAW LINE
        CMPX  #$400+32        TEST FOR END
        BNE   DRAWL           LOOP FOR ONE LINE
```

The kicker is the term "simply storing"; nothing seems to be simple in computers, with the possible exception of being able to buy new equipment. Assembly language can easily be used to draw simple horizontal and vertical lines, or even diagonal lines on the text screen, but it is quite a bit more work to draw lines of any angle or shapes on the text screen.

The Graphics Screens

The problem is solved for us on the graphics screens where we have the built-in commands of BASIC for doing all kinds of things.

The graphics modes are of two types, either a 2-color or 4-color. There are two "color sets" that can be selected by the BASIC SCREEN command, but within each color set there are only 2 or 4 colors.

There are actually 8 graphics modes available with the VDG chip; extended BASIC implements the highest resolution 5 of them, and they are, of course, selected by PMODE 0 through 4. See Figure 24-3.

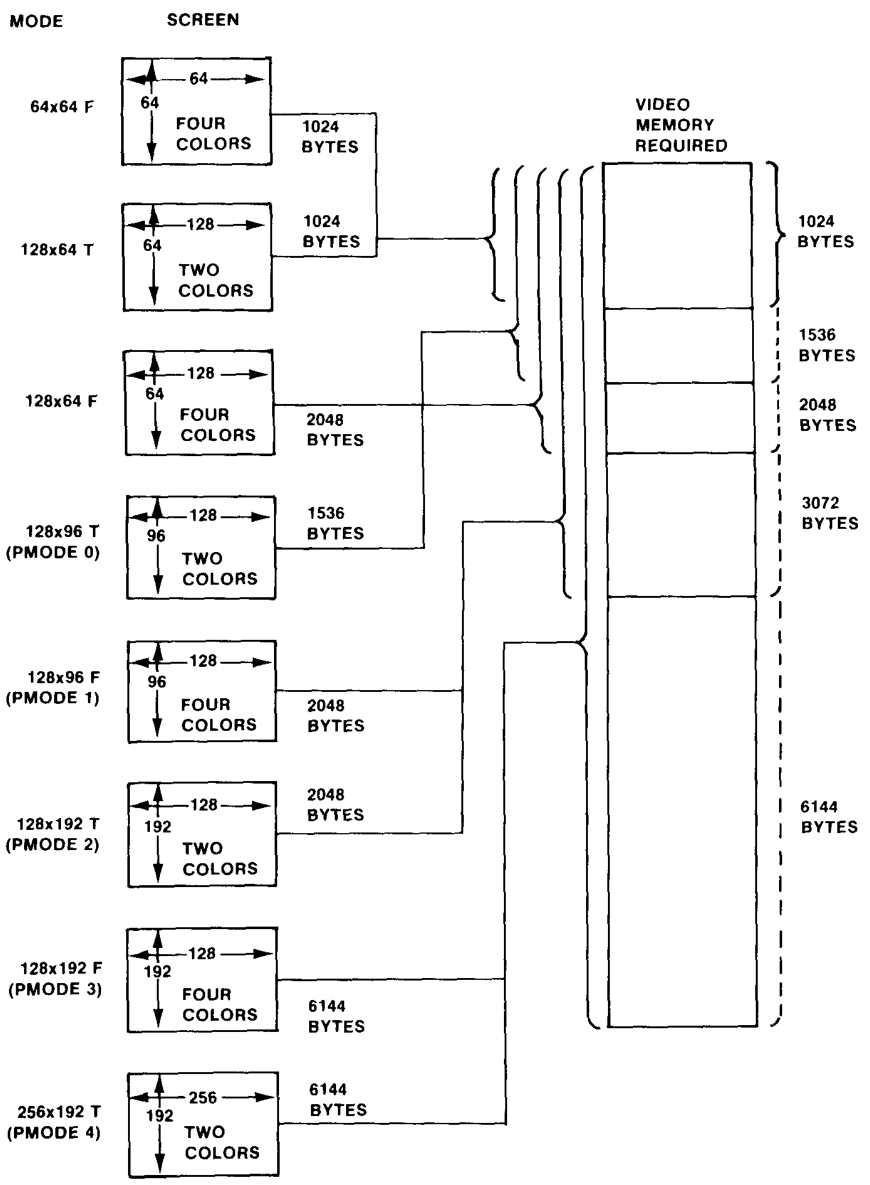


Figure 24-3. Graphics Modes

Note that there are three 2-color modes and two 4-color modes.

In the 2-color mode, there is one bit for every graphics "element" on the screen. In the highest resolution mode, for example, the 256 horizontal by 192 vertical mode (PMODE 4) there are 256*192 bits, or 49,152 bits per screen. This is equivalent to 49,152/8 or 6,144 bytes, divided up among 4 graphics pages.

In the 4-color mode, there are 2 bits for every graphics "element" on the screen. In the highest-resolution 4-color mode, for example, there's 128*192*2, or 49,152 bits per screen again.

The 2-color and 4-color mode mapping is shown in Figure 24-4. You can look on the mapping of the screen in memory as being a two-dimensional array of 32 bytes across by 192 bytes down. Each row across represents either 128 four-color elements or 256 two-color elements.

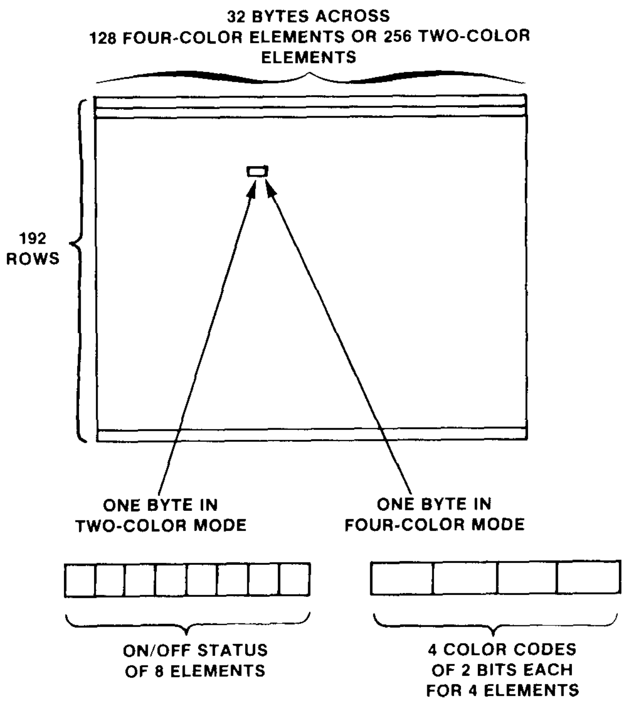

32 BYTES ACROSS
128 FOUR-COLOR ELEMENTS OR 256 TWO-COLOR ELEMENTS
192 ROWS
ONE BYTE IN TWO-COLOR MODE
ONE BYTE IN FOUR-COLOR MODE
ON/OFF STATUS OF 8 ELEMENTS
4 COLOR CODES OF 2 BITS EACH FOR 4 ELEMENTS

Figure 24-4. Two-Color and Four-Color Mapping

To set any of the elements to the foreground color in the 2-color mode, all that must be done is to set the corresponding bit to a one. In the 4-color mode, a color code of 00, 01, 10, or 11 in binary is stored in the 2 bits corresponding to the element.

Suppose that we were in 2-color PMODE 4. To set the middle row, we'd do something like:

```
        LDX    #32*96+$600        POINT TO MIDDLE ROW
        LDA    #$FF               ALL ON
LOOP    STA    ,X+                   TURN ON 8 ELEMENTS
        CMPX   #32*96+32+$600        TEST FOR ROW END
        BNE    LOOP                  GO IF NOT END
```

Of course, here again, it would be very difficult to implement a full-fledged "graphics package" set of commands, but these are in BASIC anyway. Assembly language can be used to supplement BASIC commands very nicely, however.

Hints and Kinks 24-1
Setting the Graphics Modes

BASIC sets the various graphics modes for you. However, it is possible, and not difficult, to set the graphics modes, including those "unimplemented" modes, without BASIC. You'll need to know something about the logical architecture of the Color Computer in regard to several chips in the CoCo, including the VDG. See "Color Computer Graphics."

As an example, of an assembly-language supplementary function, let's consider a program to "ZOOM" magnify a portion of a graphics screen.

The ZOOM Program

The ZOOM program magnifies or "zooms" the display, as shown in Figure 24-5. ZOOM can be called from BASIC to magnify the screen after the desired figures and graphics have been drawn. It works with one "quadrant" of the screen at a time, expanding 1/4 of the screen to full size. It operates only in PMODE 4, the 256 by 192 two-color resolution mode, although it could easily be modified to the other two 2-color modes and, without too much additional work, to the two 4-color modes.

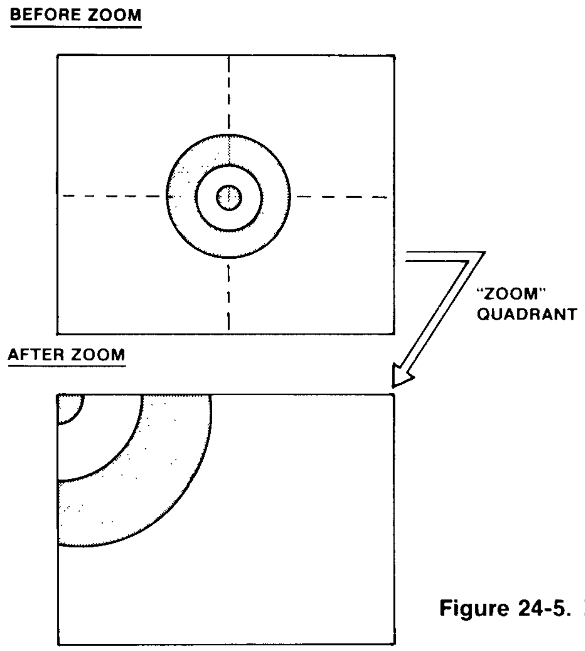


Figure 24-5. ZOOM Action

Here's how ZOOM is implemented. One of the 4 quadrants, 0 through 3, is specified. This quadrant is then physically moved to the quadrant 3 area. The operation doesn't take too long in assembly language (less than 1/10 second), and makes the calculations a lot easier. (Another version could be implemented that did not relocate the quadrant; it would have four cases.)

After the quadrant is moved to quadrant 3, one row of quadrant 3 (the "source" data) is processed at a time. Each row of the source has 256 bits. Each bit generates a cluster of 4 bits for the X4 magnification, as shown in Figure 24-6.

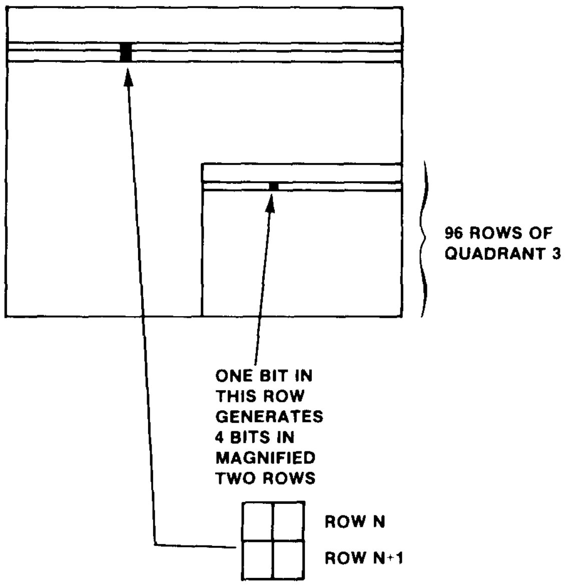


Figure 24-6. Times Four Magnification Action

The process continues through the 96 rows of the quadrant 3 data. Data is read from quadrant 3 from left to right and from top to bottom. It is stored on the full-size screen in the same fashion. By the time it gets down to the lower right-hand corner of quadrant 3, it has been written to nearly all of the screen. Reading and storing the data in this fashion avoids any problems of "overwriting" the quadrant 3 data.

The ZOOM subroutine is shown in Figure 24-7. It consists of three parts. Let's start the description from the bottom up.

```
3FØØ                          ORG     $3FØØ
                   ***********************************************
                   * X4 ZOOM OF PAGE Ø IN PMODE 4                *
                   * ENTRY: (D)=QUADRANT, Ø-3                    *
                   * EXIT:  SCREEN MAGNIFIED                     *
                   ***********************************************
          121Ø     S3QUAD     EQU     32*96+16+$6ØØ
          Ø6ØØ     SØQUAD     EQU     $6ØØ
          1EØØ     ENDLOC     EQU     $6ØØ+6144
          3FØØ     ZOOM       EQU     *
3FØØ BD   B3ED                JSR     $B3ED     GET QUADRANT
3FØ3 C4   Ø3                  ANDB    #3        DON'T TRUST CALLER'
3FØ5 86   Ø2                  LDA     #2        QTAB ENTRY SIZE
3FØ7 3D                       MUL               DISP TO QTAB ENTRY
3FØ8 C3   3F48                ADDD    #QTAB     POINT TO QTAB ENTRY
3FØB 1F   Ø3                  TFR     D,U       POINTER NOW IN U
3FØD AE   C4                  LDX     ,U        SOURCE ADDRESS
3FØF 1Ø8E 121Ø                LDY     #S3QUAD DESTINATION ADDRESS
                   * MOVE QUADRANT TO QUADRANT 3 FOR SIMPLICITY
3F13 A6   8Ø       ZOMØ1Ø     LDA     ,X+           GET BYTE
3F15 A7   AØ                  STA     ,Y+           STORE
3F17 1F   2Ø                  TFR     Y,D           DEST POINTER
3F19 C4   ØF                  ANDB    #$F           TEST 4 LS BITS
3F1B 26   F6                  BNE     ZOMØ1Ø        GO IF NOT AT 16TH
3F1D 1Ø8C 1EØØ                CMPY    #ENDLOC   TEST END
3F21 27   Ø8                  BEQ     ZOMØ2Ø    GO IF RELOCATED
3F23 3Ø   88 1Ø               LEAX    +16,X     POINT TO NXT ROW SRCE
3F26 31   A8 1Ø               LEAY    +16,Y     POINT TO NXT ROW DEST
3F29 2Ø   E8                  BRA     ZOMØ1Ø    DO FOR 96 ROWS
                   * NOW "X4" FROM QUADRANT 3
3F2B 8E   121Ø     ZOMØ2Ø     LDX     #S3QUAD START OF QUAD 3
3F2E 1Ø8E Ø6ØØ                LDY     #SØQUAD START OF QUAD Ø
3F32 8D   1C       ZOMØ3Ø     BSR     STRIPB        1-4 BIT CONV, 8 TIMES
3F34 1F   1Ø                  TFR     X,D           SOURCE PNTR
3F36 C4   ØF                  ANDB    #$F           TEST 4 LS BITS
3F38 26   F8                  BNE     ZOMØ3Ø        GO IF NOT AT 16TH
3F3A 8C   1EØØ                CMPX    #ENDLOC   TEST END
3F3D 27   Ø8                  BEQ     ZOMØ9Ø    GO IF DONE
3F3F 3Ø   88 1Ø               LEAX    +16,X     POINT TO NEXT SOURCE ROW
3F42 31   A8 2Ø               LEAY    +32,Y     POINT TO NEXT DEST ROW
3F45 2Ø   EB                  BRA     ZOMØ3Ø    CONTINUE
3F47 39            ZOMØ9Ø     RTS             RETURN
3F48      Ø6ØØ     QTAB       FDB     SØQUAD          QUAD Ø
3F4A      Ø61Ø                FDB     16+$6ØØ         QUAD 1
3F4C      12ØØ                FDB     32*96+$6ØØ      QUAD 2
3F4E      121Ø                FDB     S3QUAD          QUAD 3
                   ***********************************************
                   * CONVERT 1 BIT TO 4 BITS, 8 TIMES            *
                   * ENTRY: X=>SOURCE                            *
                   *        Y=>DESTINATION, UPPER LEFT CORNER    *
                   * EXIT:  32 BITS UPDATED                      *
                   ***********************************************
3F5Ø C6   Ø8       STRIPB     LDB     #8        LOOP COUNT
3F52 6F   A4                  CLR     ,Y        CLEAR NEW
3F54 6F   21                  CLR     +1,Y
3F56 6F   A8 2Ø               CLR     +32,Y
3F59 6F   A8 21               CLR     +33,Y
3F5C A6   84                  LDA     ,X        GET SOURCE BYTE
3F5E 44            STRØ2Ø     LSRA                  SHIFT OUT BIT
3F5F 24   ØF                  BCC     STRØ3Ø    GO IF Ø BIT
3F61 6C   21                  INC     +1,Y      SET UPPER ROW 2 BITS
3F63 6C   21                  INC     +1,Y
3F65 6C   21                  INC     +1,Y
3F67 6C   A8 21               INC     +33,Y     SET LOWER ROW 2 BITS
3F6A 6C   A8 21               INC     +33,Y
3F6D 6C   A8 21               INC     +33,Y
3F7Ø 33   A4       STRØ3Ø     LEAU    ,Y        UPPER ROW ADDRESS
```

Figure 24-7. ZOOM Program

```
3F72 34   02             PSHS    A          SAVE SOURCE BYTE
3F74 8D   0F             BSR     SHIFT      RIGHT ROTATE
3F76 33   A8 20          LEAU    +32,Y      LOWER ROW ADDRESS
3F79 8D   0A             BSR     SHIFT      RIGHT ROTATE
3F7B 35   02             PULS    A          RETRIEVE SOURCE BYTE
3F7D 5A                  DECB               DECREMENT COUNT
3F7E 26   DE             BNE     STR020     GO IF NOT 8
3F80 30   01             LEAX    +1,X       BUMP SOURCE
3F82 31   22             LEAY    +2,Y       BUMP DESTINATION
3F84 39                  RTS              RETURN
                   **********************************************
                   * RIGHT ROTATE 16 BITS POINTED TO BY U       *
                   **********************************************
3F85 A6   C4       SHIFT  LDA'    ,U      GET MS BYTE
3F87 44                   LSRA            GET LS BIT IN C
3F88 66   41              ROR     +1,U    ROTATE TO LS BYTE
3F8A 66   C4              ROR     ,U      ROTATE MS BYTE
3F8C 66   41              ROR     +1,U    ROTATE LS BYTE
3F8E 66   C4              ROR     ,U      ROTATE MS BYTE
3F90 39                   RTS             RETURN
          0000            END
00000 TOTAL ERRORS

ENDLOC   1E00
QTAB     3F48
S0QUAD   0600
S3QUAD   1210
SHIFT    3F85
STR020   3F5E
STR030   3F70
STRIPB   3F50
ZOM010   3F13
ZOM020   3F2B
ZOM030   3F32
ZOM090   3F47
ZOOM     3F00
```

Figure 24-7 continued

The SHIFT Subroutine

The SHIFT subroutine does two right rotates on 16 bits of graphics memory. This is two adjacent bytes.

The subroutine is entered with the U register pointing to the most significant byte to be shifted. This byte is loaded into A and shifted left by the LSRA, setting the Carry to a 1 or 0. This is a "dummy" shift, done only to set the Carry.

The following ROR rotates the two bytes at 0,U and +1,U to the right. Data shifted out of the least significant byte is shifted back into bit 7 of the most significant byte by the ROR +1,U.

The STRIPB Subroutine

This subroutine takes one byte from the source row of quadrant 3 and generates 64 bits on the full-sized screen. In doing so, it goes through 8 "iterations," one for each bit of the source byte.

The subroutine is entered with the X register pointing to the source byte in quadrant 3 and with Y pointing to the "destination" byte on the full-size screen.

First, B is loaded with 8, to count the 8 iterations. Next, the four bytes on the full-sized screen are cleared. If the source byte contains all zeroes, the operation is done, but there will probably be 1 or more "1" bits in the source byte. The CLR is done to ,Y (the 8 upper-left elements), +1,Y (the 8 upper-right element), +32,Y (the 8 lower-left elements), and +33,Y (the 8 lower-right elements).

The source byte is now loaded into A.

STR020 through the remainder of the subroutine is an 8-iteration loop. Each time through the loop, the following actions occur:

- A bit is shifted right out of A into the Carry.
- If this bit is a 1 (CS), a "11" is loaded into the two least significant bits of each destination row. These bits will be right rotated 8 times.
- At STR030, U is loaded with the upper row address from Y and SHIFT is called to rotate the upper row. SHIFT is called again to rotate the lower row (U=+32,Y).
- The A register was saved in the S stack before the BSRs and is restored by the PULS A.
- The count in B is then decremented. If not 8 iterations, another pass is made.
- After the 8th iteration, the source pointer in X is incremented by one (8 source row bits have been processed), and the destination pointer is incremented by 2 (16 destination row bits have been processed).

Main Line Code

The "main line" code is at ZOOM.

We've used three EQUates to preset three values. S3QUAD is the location of the third quadrant starting byte. S0QUAD is the location of the first quadrant starting byte. ENDLOC is one more than the last byte location in the graphics screen.

ZOOM first calls the $B3ED ROM subroutine to get the quadrant number. This is returned in D. The B register is then ANDed with 3 to get a number of 0-3 (not that we don't trust the caller).

The result of 0-3 is then MULtiplied by 2 to yield 0 through 6. This "index value" is then added to "QTAB", the starting address of a 4-entry table that defines the starting byte address for each of the 4 quadrants.

D now points to one of the 4 starting addresses in QTAB. The contents of D is transferred to U, and U is used to load the starting address into the X register. Y is loaded with the starting address of quadrant 3.

The ZOM010 loop moves quadrant 0 through 3 into quadrant 3. Note that quadrant 3 may be moved into quadrant 3, resulting in no change to the screen. In the loop, X points to the source (original quadrant) and Y points to the destination (quadrant 3). Auto increment is used on both. A check is made for the end of each row by ANDing the Y pointer value with $F. The result will only be 0000 at the end of each row. The end of quadrant 3 is reached at the end of the transfer when Y contains the ENDLOC value.

At ZOM020, the original quadrant is now in quadrant 3. The data in quadrant 3 will now be transferred one row at a time to the full-sized screen. During the loop of 96 iterations, X contains the pointer to the next byte of quadrant 3, and Y contains the pointer to the upper-left byte of the 4-byte full-sized destination. Each byte of the quadrant 3 data is used in STRIPB. An end of row check of X is made by ANDB #$F. At the end of each quadrant 3 row, X is incremented by 16 to bypass the quadrant 2 portion of the row, and Y is incremented by 32 to **bypass the next destination row, which has already been filled with data.** The full size data storage proceeds two rows at a time for every row of the quadrant 3 data.

Hints and Kinks 24-2
Changing ZOOM to Four-Color Operation

To change ZOOM to 4-color operation, change STRIPB to look at four sets of 2 bits (4 iterations). Don't CLR the 32 bits of the destination beforehand, but write out the 2-bit pattern four times. You can still use SHIFT as is, and the main-line code in ZOOM should also be intact. (I know, if it was so easy, why didn't I do it. . .)

Hints and Kinks 24-3
The INCs in ZOOM

The three INC +1,Y and three INC +33,Y instructions effectively set three bits in either the upper or lower row. Don't forget that the four bytes in the full-sized screen were cleared initially. Incrementing three times is the same as ANDing in a binary 11, in this case. We can't do an AND to a memory byte.

Using ZOOM

The best way to use ZOOM is to assemble onto cassette tape without using the "/IM" option (use /AO, however, for the ORG). After doing a CLEAR 200,&H3EFF in BASIC, load the ZOOM object with the CLOADM command in BASIC. Don't do an EXEC, of course.

To call ZOOM, do a

```
DEFUSR0=&H3F00
A=USR0(N)
```

where N is the quadrant number of 0 through 3. The DEFUSR0 needs to be done only once, but the USR0 call can be done anytime a magnification of a quadrant is required.

Hints and Kinks 24-4
ZOOM Aesthetics

Although the transfer of the original quadrant to quadrant 3 doesn't take long, it doesn't look "clean." Consider copying the current screen to the remaining 4 pages and doing the operation "off line"; you'll have to change the $600 references to $1E00 to work with pages 5 through 8.

Review

To review this chapter:

- The text screen starts at $400 and the graphics pages at $600
- The text screen has 512 character positions which may be an alphanumeric character or a graphics character
- Mapping for the text screen is done one byte per character position; the msb of each byte determines alphanumeric or graphics character
- Graphics mode permits 2- or 4-color graphics displays of up to 256- by 192-element resolution
- Mapping for the graphics pages is done on a bit-by-bit basis for 2-color mode, or two bits per element for the 4-color mode

For Further Study

"Color Computer Graphics" (if you'd just buy the blasted book, I'd stop touting it. . .)

KEY CHART — CHAPTER 25

INSTRUCTIONS

~~ABX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~ADCA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~ADCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~ADDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~ADDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~ADDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~ANDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~ANDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~ANDCC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~ASLA~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~ASLB~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
~~ASL~~	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~ASRA~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~ASRB~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
~~ASR~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~BCC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~LBCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~BCS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~LBCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~BEQ~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~LBEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~BGE~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~LBGE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~BGT~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~LBGT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~BHI~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~LBHI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~BHS~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

ADDRESSING MODES

~~INHERENT~~
~~DIRECT~~
~~EXTENDED~~
~~IMMEDIATE~~
~~SIMPLE INDEXED~~
~~RELATIVE~~
~~DISPLACEMENT INDEXED~~
~~AUTO INCREMENT/DECREMENT~~
~~INDIRECT~~
~~SOPHISTICATED~~

PSEUDO OPS

~~EQU~~	~~ORG~~
~~FCB~~	~~RMB~~
~~FCC~~	~~SET~~
~~FDB~~	~~SETDP~~

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TA~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERRO~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLO~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOIN~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDI~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	~~→ BRANCH INDIRECT~~
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,B~~
~~P SAVE ML ON TAPE~~	~~: FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETI~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	~~PASSING PARAMETER~~
~~INCREMENTS/DECREMENTS~~	~~VARPTR USE~~
~~COMPLEMENTS~~	~~ROM SUBROUTINES~~
~~LOGICAL OPERATIONS~~	~~OTHER ADDRESSING~~
~~MULTIPLE PRECISION~~	~~GRAPHICS~~
~~DATA VALUES~~	**SOUND**
~~INDEXING~~	LARGER PROGRAMS
~~INDEXING WITH X,Y~~	
~~SORTING~~	

Bold Type Present Chapter
Regular Type Future Chapters
~~*Italic Type*~~ · *Past Chapters*

Chapter 25
Assembly-Language Sound

In this chapter we'll look at the mysteries of generating sound in the Color Computer. Sound must be generated by assembly-language programming, whether that programming is in the BASIC interpreter or in your own program. Even though BASIC has quite a powerful sound and music capability, there's a lot more that can be done via assembly language.

How Sound is Generated

The Digital-to-Analog Converter

A Digital-to-Analog Converter is a device for converting digital values found in a computer to "analog" voltage levels.

A digital voltage level is either a one or a zero, an on or an off, and that translates into voltage levels of about +5 volts or 0 volts in the Color Computer and many other microcomputers, a little less than the voltage level of a lantern battery.

An analog voltage level, however, can be any value between the 0 volts and +5 volts — 3.5, 4.7, 2.3 volts, or other values.

The Color Computer has a built-in digital-to-analog converter, shown in Figure 25-1. We'll call it a "DAC" for short.

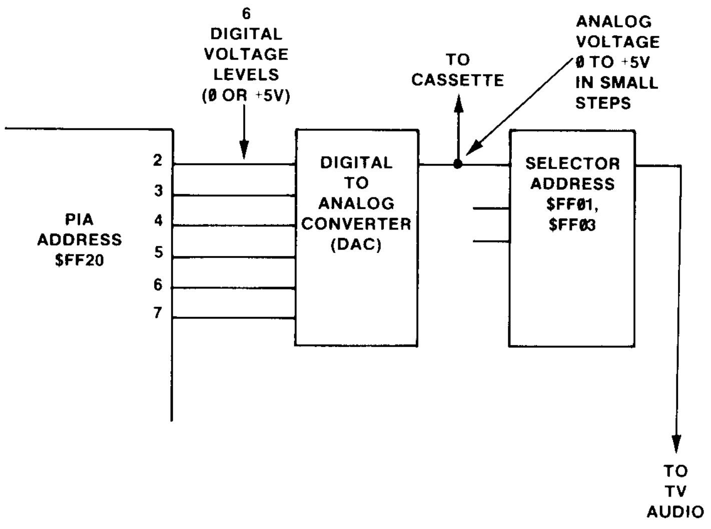

6 DIGITAL VOLTAGE LEVELS (0 OR +5V)
TO CASSETTE
ANALOG VOLTAGE 0 TO +5V IN SMALL STEPS
PIA ADDRESS $FF20
2
3
4
5
6
7
DIGITAL TO ANALOG CONVERTER (DAC)
SELECTOR ADDRESS $FF01, $FF03
TO TV AUDIO

Figure 25-1. Color Computer Digital-to-Analog Convertor

The DAC takes a 6-bit value and converts it to a voltage level of 0 through about +5 volts. The higher the 6-bit value, the greater the voltage will be. A digital value of 6 bits can hold 64 discrete levels, 000000 through 111111, so the "step" between levels is about 5/64 volts, or approximately 0.08 volts. A digital value of 000001 will generate 0.08 volts, a value of 000010, 0.15 volts, a value of 100000, 2.5 volts, and so forth. The conversion steps are shown in Figure 25-2.

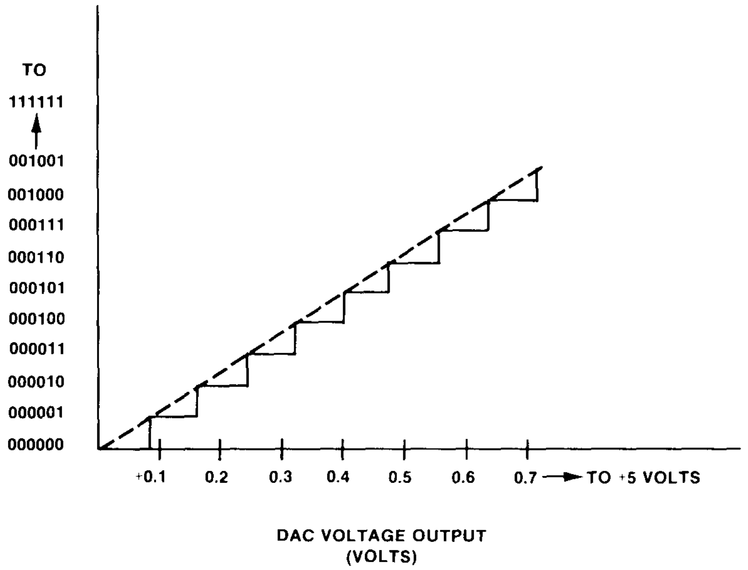


Figure 25-2. DAC Conversion Steps

The DAC is connected to a device called a "PIA," for Peripheral Interface Adapter. This is a semiconductor chip that contains several sets of "output lines." The address of the PIA for the DAC is $FF20, and it is addressed just like any memory location. Once a value is written to this PIA, it remains in the PIA and appears on the output lines as digital voltages of +5 or 0 volts, ones or zeroes. To store a new value in the PIA, another write is done. The writes are Store instructions, just as you would store data in memory.

The output lines of the PIA are connected to the DAC. The DAC takes the 6 PIA outputs and converts them to an analog voltage, which appears at the DAC output as a single line, just as you'd have a single line for an audio amplifier speaker connection (really two wires, but one is "ground").

The DAC output is routed to another device which selects one of three inputs and routes it to the audio portion of your television receiver. The output of the DAC is also routed to the cassette tape output.

Hints and Kinks 25-1
The ADC

The Color Computer also has an analog-to-digital converter, an "ADC" which performs the reverse function of a DAC. It converts an analog voltage into a 6-bit digital value from 0 through 63. The

ADC is used primarily in the joystick input conversion. The joystick position is converted to a resistance value by the joystick "pot," which produces a voltage value from +0 to +5 volts. This voltage is then converted to a digital value of 000000 through 111111 for both the x and y axes. The ADC conversion is done primarily in "software."

Generating a Sine Wave

A pure sine wave looks like Figure 25-3. We've all heard them as test tones on television or fm stations. The DAC circuitry can be used to generate a reasonable facsimile of a sine wave. As a matter of fact, this is **exactly** what is done to write data on cassette tape for BASIC, EDTASM+, and other programs — a sine wave is "synthesized" by the DAC from digital values. Here's how it works:

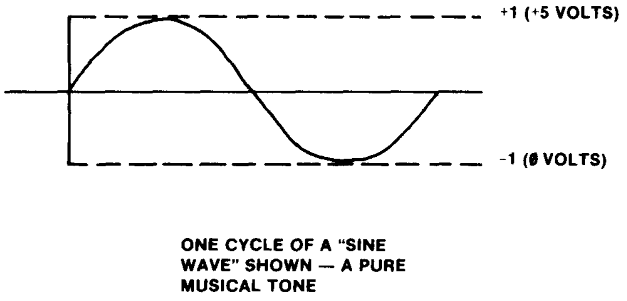


Figure 25-3. Sine Wave

- A sine wave table is constructed in memory. The sine wave table in BASIC is at location $A85C (this location may change in subsequent ROM versions) and consists of 36 digital values as shown in Figure 25-4.

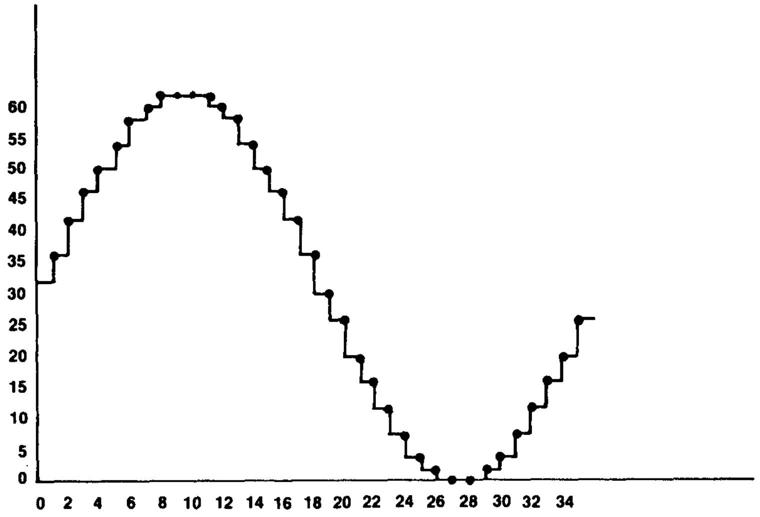


Figure 25-4. DAC Sine Wave Generation

Hints and Kinks 25-2
Sine Wave Generation

As you can see from the figure, the sine wave is actually a "stepped wave" consisting of small increments or steps. It's good enough for cassette recording purposes. Any DAC will create a wave of this type; the more bits, the finer the step will be.

- The table values are written to PIA location $FF20, one at a time.
- After each value is written, a time delay occurs. The length of this time delay determines the "frequency" of the tone.
- The next value of the table is accessed and written. If the end of the sine wave table is reached, the table pointer is reset to the beginning.
- A preset number of "cycles" of the table is written out over the duration of the tone.

The output frequencies used for cassette are 1200 and 2400 hertz, or cycles per second. Each "cycle" of the sine wave produced will be 0.00083 or 0.000416 seconds long, and in that time we can execute several hundreds of 6809 instructions, making the process an easy task for assembly-language code.

Generating Other Sounds
The sine wave is a "pure" sound. Other sounds are much more complex, containing many different frequency components, as shown in Figure 25-5.

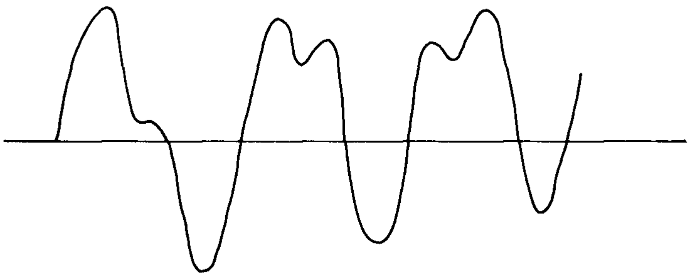


Figure 25-5. Complex Sounds

It's easy to generate any "repeatable" sound by using the "table" approach to generate a continuous number of cycles of the sound, just as is done in the sine wave case. Here's a short program that will generate a burst of sound over n cycles.

```
                   3F07              00100        ORG     $3F07
                   00110 ***********************************************
                   00120 * SOUND GENERATOR FROM MEMORY                 *
                   00130 ***********************************************
3F07 B6   FF01     00140 SOUND    LDA     $FF01     SELECT SOUND OUT
3F0A 84   F7       00150          ANDA    #$F7      RESET MUX BIT
3F0C B7   FF01     00160          STA     $FF01     STORE
3F0F B6   FF03     00170          LDA     $FF03     SELECT SOUND OUT
3F12 84   F7       00180          ANDA    #$F7      RESET MUX BIT
3F14 B7   FF03     00190          STA     $FF03     STORE
3F17 B6   FF23     00200          LDA     $FF23     GET PIA
3F1A 8A   08       00210          ORA     #8        GET 6-BIT SOUND ENABLE
3F1C B7   FF23     00220          STA     $FF23     STORE
3F1F 10BE 3F00     00230          LDY     $3F00     GET # OF TIMES
3F23 BE   3F03     00240 SND010   LDX     $3F03       GET START ADDRESS
3F26 A6   80       00250 SND020   LDA     ,X+           GET NEXT BYTE
3F28 84   FC       00260          ANDA    #$FC          RESET 2 LS BITS
3F2A B7   FF20     00270          STA     $FF20         OUTPUT
3F2D 8D   0A       00280          BSR     DELAY         DELAY
3F2F BC   3F05     00290          CMPX    $3F05         TEST FOR END
3F32 26   F2       00300          BNE     SND020        GO IF NOT END
3F34 31   3F       00310          LEAY    -1,Y        DECREMENT # OF TIMES
3F36 26   EB       00320          BNE     SND010        GO IF NOT 0
3F38 39            00330          RTS               RETURN - DONE
3F39 B6   3F02     00340 DELAY    LDA     $3F02     GET DELAY COUNT
3F3C 4A            00350 DEL010   DECA                COUNT-1
3F3D 26   FD       00360          BNE     DEL010      GO IF NOT 0
3F3F 39            00370          RTS               RETURN
          0000     00380          END
00000 TOTAL ERRORS

DEL010   3F3C
DELAY    3F39
SND010   3F23
SND020   3F26
SOUND    3F07
```

Figure 25-6. SOUND Program

The SOUND program uses four parameters from the $3F00 area, as shown in Figure 25-7. The pattern for the sound is taken from a table defined by the start address at $3F03,4, and an end address defined by the contents of $3F05,6. The delay between outputs to the DAC is defined by the contents of $3F02. This is a value of $01 through $FF or $00; longer delays cause lower sounds; a value of $00 will be the longest delay. The contents of $3F00,1 is a "repeat count." This is a count of $0001 through $FFFF or $0000 which defines the number of times the pattern will be repeated ($0000 is a repeat count of 65,536).

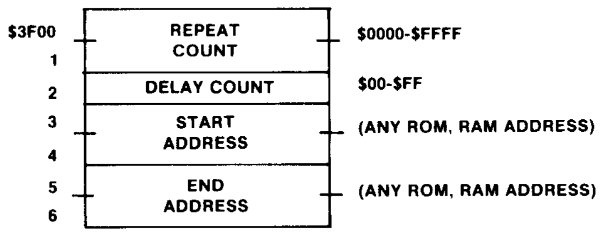


Figure 25-7. SOUND Parameter Block

Here's how the SOUND program works:

The DAC output is first routed to the television audio by resetting bit 3 of address $FF01 and bit 3 of address $FF03. These addresses are "PIA" addresses that control the routing of the sound to the tv audio.

Next, bit 3 of address $FF23 is set. Again, this is a PIA address that among other things, controls the "6-bit sound enable" by bit 3. Setting bit 3 turns on the DAC sound.

The main program itself consists of two loops, one nested inside the other. Let's look at the innermost loop first, from SND020 through the BNE SND020.

This loop loads the next byte from the table in memory, using X as a pointer. The two least significant bits of the value are then reset by an AND #$FC. The lower two bits of the PIA are used for other functions (cassette and serial I/O) and this operation stores zeroes in the two least significant bits.

The value is then stored in PIA $FF20 by the STA $FF20 instruction, causing an almost instantaneous DAC output.

The DELAY subroutine is then called. DELAY gets the delay count from location $3F02 and decrements it until it reaches 0.

After the delay, the X register is compared to the end address in $3F05. If the end address has not been reached, another value is picked up from the table by looping back to SND020.

This inner loop outputs the entire table from beginning to end with the delay between values from $3F02.

The outer loop starts at SND010 and continues through the BNE SND010. It decrements the "repeat count" in Y, looping back to another cycle of the inner loop if the repeat count is not zero.

X is initialized with the starting address and Y with the repeat count at the beginning of the program.

A BASIC program containing the SOUND program as embedded DATA values is shown below. It relocates the code for SOUND to the $3F07 area. All instructions in SOUND are position independent (relocatable) so that it can be moved anywhere in memory, as long as the variable area at $3F00 through $3F06 is maintained.

```
100 ' BASIC DRIVER FOR SOUND
101 CLEAR 200,&H3EFF
110 DATA 182,255,1,132,247,183,255,1,182,255,3
120 DATA 132,247,183,255,3,182,255,35,138,8,183
130 DATA 255,35,16,190,63,0,190,63,3,166,128,132,252,183
140 DATA 255,32,141,10,188,63,5,38,242,49,63,38,235,57
150 DATA 182,63,2,74,38,253,57
160 FOR I=&H3F07 TO &H3F07+56
170 READ A: POKE I,A
180 NEXT I
190 DEFUSR0=&H3F07
200 INPUT "#TIMES,DELAY,ST,END";N,D,S,E
210 POKE &H3F00,INT(N/256): POKE &H3F01,N-INT(N/256)*256
220 POKE &H3F02,D
230 POKE &H3F03,INT(S/256): POKE &H3F04,S-INT(S/256)*256
240 POKE &H3F05,INT(E/256): POKE &H3F06,E-INT(E/256)*256
250 A=USR0(0)
260 GOTO 200
```

Figure 25-8. BASIC SOUND Driver Program

This is a fun program to run! You might try the sine wave table at $A85C first, to see how the program works. Try any other tables in memory, ROM or RAM, with different delays and repeat counts. You can generate almost an unlimited number of interesting sounds including phasor blasts, whistles, explosions, and crashes.

Hints and Kinks 25-3
Game Sounds

After running this program, it's easy to see where some of those arcade game sounds come from — from instructions and data in ROM memory! Just choose a suitable sounding set of instructions. You never thought you'd be able to **hear** your assembly-language code, did you?

A Music Synthesizer

The SOUND program above is one approach to generating sounds. Another approach is shown in Figure 25-9. This is a "music synthesizer" implemented in software.

```
3FØØ                              ORG      $3FØØ
                         ******************************************
                         * MUSIC SYNTHESIZER                      *
                         * ENTRY: ($3FFØ)=FREQ DELAY COUNT        *
                         *        ($3FF1,2)=ENVELOPE TABLE ADDRESS *
                         *        ($3FF3,4)=ENVELOPE DELAY COUNT  *
                         *        ($3FF5)=VOLUME, 1 TO 255        *
                         ******************************************
3FØØ B6   FFØ1           MUSSYN   LDA      $FFØ1    SELECT SOUND OUT
3FØ3 84   F7                      ANDA     #$F7     RESET MUX BIT
3FØ5 B7   FFØ1                    STA      $FFØ1    STORE
3FØ8 B6   FFØ3                    LDA      $FFØ3    SELECT SOUND OUT
3FØB 84   F7                      ANDA     #$F7     RESET MUX BIT
3FØD B7   FFØ3                    STA      $FFØ3    STORE
3F1Ø B6   FF23                    LDA      $FF23    GET PIA
3F13 8A   Ø8                      ORA      #8       SET 6-BIT SOUND ENABLE
3F15 B7   FF23                    STA      $FF23    STORE
3F18 CE   3FFØ                    LDU      #$3FFØ   POINT TO BLOCK
3F1B AE   41                      LDX      +1,U     GET ENVELOPE ADDRESS
3F1D BF   3F69                    STX      ENVPTR   STORE IN PNTR
3F2Ø AE   43                      LDX      +3,U     GET ENV DELAY
3F22 A6   9F 3F69        MUSØØ5   LDA      ↑ENVPTR←   GET VALUE
3F26 27   4Ø                      BEQ      MUSØ9Ø     IF Ø, DONE
3F28 E6   45                      LDB      +5,U       GET VOLUME
3F2A 3D                           MUL                 FIND VALUE
3F2B 84   FC                      ANDA     #$FC       RESET RS-232-C
3F2D B7   FF2Ø                    STA      $FF2Ø      SET ON
3F3Ø E6   C4                      LDB      ,U         GET FREQ DELAY COUNT
3F32 3Ø   1F             MUSØ1Ø   LEAX     -1,X         DECRE ENV COUNT
3F34 26   ØC                      BNE      MUSØ2Ø       GO IF NOT Ø
3F36 1ØBE 3F69                    LDY      ENVPTR       GET ENV PNTR
3F3A 31   21                      LEAY     1,Y          NEXT SEGMENT
3F3C 1ØBF 3F69                    STY      ENVPTR       STORE
3F4Ø AE   43                      LDX      +3,U         GET ENVEL DELAY
3F42 5A                  MUSØ2Ø   DECB                  DECREMENT F COUNT
3F43 26   ED                      BNE      MUSØ1Ø       GO IF NOT Ø
3F45 A6   9F 3F69                 LDA      ↑ENVPTR←   DUMMY
3F49 21   ØØ                      BRN      *+2        DUMMY
3F4B E6   45                      LDB      +5,U       DUMMY
3F4D 3D                           MUL                 DUMMY
3F4E 7F   FF2Ø                    CLR      $FF2Ø      SET OFF
3F51 E6   C4                      LDB      ,U         GET FREQ DELAY
3F53 3Ø   1F             MUSØ3Ø   LEAX     -1,X         DECRE ENV COUNT
3F55 26   ØC                      BNE      MUSØ4Ø       GO IF NOT Ø
3F57 1ØBE 3F69                    LDY      ENVPTR       GET ENV PNTR
3F5B 31   21                      LEAY     1,Y          NEXT SEGMENT
3F5D 1ØBF 3F69                    STY      ENVPTR       STORE
3F61 AE   43                      LDX      +3,U         GET ENVEL DELAY
3F63 5A                  MUSØ4Ø   DECB                  DECREMENT F COUNT
3F64 26   ED                      BNE      MUSØ3Ø       GO IF NOT Ø
3F66 2Ø   BA                      BRA      MUSØØ5     LOOP
3F68 39                  MUSØ9Ø   RTS               RETURN
3F69      ØØØØ           ENVPTR   FDB      Ø        POINTS TO ENVELOPE TABLE
          ØØØØ                    END
ØØØØØ TOTAL ERRORS

ENVPTR   3F69
MUSØØ5   3F22
MUSØ1Ø   3F32
MUSØ2Ø   3F42
MUSØ3Ø   3F53
MUSØ4Ø   3F63
MUSØ9Ø   3F68
MUSSYN   3FØØ
```

Figure 25-9. MUSSYN Program

Hardware music synthesizers generate a sine wave, square wave, or triangular wave or other waveshape which is then "modulated" by a second waveshape, and "filtered" to remove a certain portion of high frequencies (see Figure 25-10). Various other characteristics of the sound output can be changed to produce electronic music.

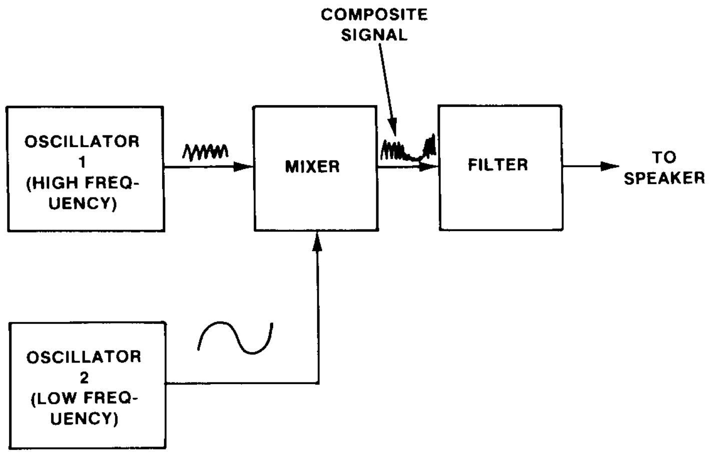


Figure 25-10. Hardware Music Synthesizers

This software approach to a synthesizer doesn't give you the flexability of a hardware synthesizer, but it does produce some very unique sounds, and you can easily modify the characteristics and volume of the sounds to create vibrato, bell tones, and other sounds that defy description.

The heart of MUSSYN is a square-wave generator. It produces a square wave with a programmable frequency. We've used a square wave because it's easy to generate, but also because it's an interesting sound. There are a lot of "odd harmonics" in a square wave, which makes for a "thick" sound, similar to a lot of hardware synthesizer output.

This basic square-wave frequency is modified by two parameters, a "volume" parameter which controls the intensity of the sound, and an "envelope" table, which controls the intensity of the sound over time, as shown in Figure 25-11.

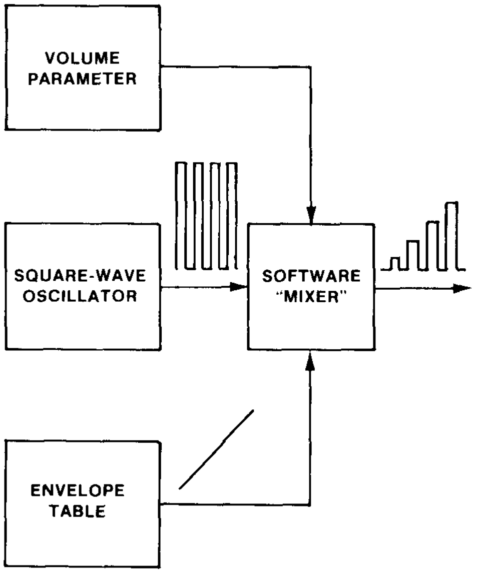


Figure 25-11. MUSSYN Parameters

Let's look at how the program operates.

Parameters

The program is "driven" by 4 parameters contained in a parameter block at location $3FF0 (see Figure 25-12).

$3FFØ	FREQUENCY DELAY CNT	1-255 (Ø 256)
1 2	ENVELOPE TABLE ADDRESS	POINTS TO WAVE-SHAPE TABLE
3 4	ENVELOPE DELAY COUNT	1-65535 (Ø= 65536)
5	VOLUME	1-255 (Ø NOT ALLOWED UNLESS YOU HATE MUSIC)

Figure 25-12. MUSSYN Parameter Block

The first parameter is a frequency delay count, which determines the frequency of the square wave by adjusting the time delay between outputs to the DAC.

The second parameter is the address of an "envelope table." This table is a table of 8-bit values from 1 to 255 which determine the magnitude of the DAC output. The delay between new envelope values is determined by the envelope delay count found in the third parameter. The envelope table and delay count determine the waveshape of the tone and create the timbre of the sound — whether it sounds like a bell, harpsichord, etc.

The fourth parameter is the volume. This value is multiplied times the envelope values to reduce the overall volume of the sound, but to keep the same shape.

Program Operation

First of all, the DAC sound output is routed to the tv audio as before, by outputting to the PIAs at locations $FF01, $FF03, and $FF23.

The U register is now loaded with the address of the parameter block. U will maintain this address throughout the subroutine.

The envelope address is now stored in location "ENVPTR." Variable ENVPTR will be incremented periodically, based on the envelope delay value, to point to the next byte of the envelope. A byte of 0 terminates the envelope and causes an RTS to the calling program.

Next, the envelope delay is put into the X register. X will hold this delay throughout the subroutine, and it will be decremented down to 0 for each envelope value.

The main loop is at MUS005 and continues through one instruction before the MUS090 label. The loop continues for each envelope value and terminates when an envelope value of 0 is reached. Each time through the loop, the following actions occur:

- (MUS005): The ENVPTR value is obtained by an indirect load of ENVPTR. If this value is 0, an RTS is done.
- This value is multiplied by the volume value (+5,U). The high-order result in A is ANDed with $FC to reset the 2 lower bits, as they are not DAC bits.
- The value is output to the PIA at $FF20, "turning on" the top of the square wave.
- The frequency delay count is put into the B register.
- (MUS010): An inner loop from MUS010 through MUS020 plus one instruction decrements the delay count in B. At the same time it decrements the envelope count in X. If the envelope count reaches zero, the ENVPTR variable is incremented to point to the next table value, and X is reinitialized with the envelope delay count. Note that the envelope and frequency delay count are both adjusted independently.
- When the frequency count is decremented down to 0, the second loop at MUS030 through MUS040 plus one instruction is executed. This loop outputs a 0 to the PIA, zeroing the DAC output (CLR $FF20) and creating the bottom edge of the square wave. It then delays as in the previous loop. Dummy instructions are done to keep the "width" of the square wave bottom approximately equal to the square wave top.

Hints and Kinks 25-4
50% Duty Cycle Square Waves

A square wave with equal top and bottom widths is called "50% duty cycle." There's really no reason to make the tops and bottoms the same width. If you vary the duty cycle, you'll get yet another set of sounds, although the effect won't be as dramatic as varying the wave envelope.

- Throughout both loops, X is decremented down to 0. Every time it reaches 0, a new envelope table value is pointed to by ENVPTR. The envelope delay is approximately constant for any frequency delay because the two loops are continually active.

A BASIC Driver for MUSSYN

Figure 25-13 shows a BASIC driver for MUSSYN. This driver sets up the parameter block at $3FF0, except for the envelope table address and the frequency delay count. It then generates four sets of sounds, shown in Figure 25-14.

```
100 'DRIVER FOR MUSSYN
110 DATA 12,24,36,48,60,72,84,96,108,120,132,144,156
120 DATA 168,180,192,204,216,228,0
130 DATA 228,216,204,192,180,168,156,144,132,120,108
140 DATA 96,84,72,60,48,36,24,12,0
150 DATA 32,96,160,240,240,240,240,240,160,128,96,80,60
160 DATA 47,32,20,16,8,8,0
170 DATA 240,216,192,176,168,168,176,192,216,240,216
180 DATA 192,176,168,168,176,192,216,240,0
190 DEFUSR=&H3F00
200 POKE &H3FF3,8:POKE &H3FF4,0
210 POKE &H3FF5,255
220 FOR E=0 TO 3
230 POKE &H3FF1,&H3B+E
240 POKE &H3FF2,0
250 FOR L=&H3B00+E*256 TO &H3B00+E*256+19
260 READ A:POKE L,A
270 NEXT L
280 FOR F=10 TO 250 STEP 10
290 POKE &H3FF0,F
300 A=USR0(0)
310 FOR I=0 TO 100:NEXT I
320 NEXT F
330 NEXT E
```

Figure 25-13. BASIC MUSSYN Driver Program

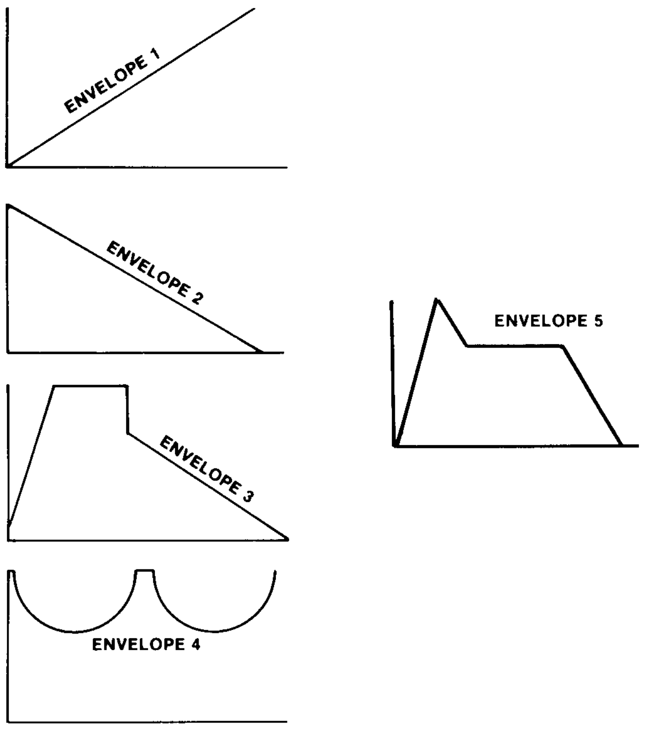


Figure 25-14. Envelopes for MUSSYN

Be certain to protect memory by a CLEAR 200, &H3EFF before loading the object of MUSSYN or running the BASIC driver.

The first sound is a rising intensity sound that sounds unlike any musical tone you've heard. It's like a tape recording played backwards.

The second sound sounds like a bell that's rung but one that doesn't stay "ringing" for any period of time.

The third sound sounds vaguely like a wind instrument blown for short bursts.

The fourth sound is a kind of vibrato sound on an electronic organ.

You can easily build your own envelope tables by entering values. Remember to make the terminating entry a zero. Experiment especially with the fifth waveshape shown in the figure above (unimplemented). This "ADSR" (attack, decay, sustain, release) is characteristic of many musical instruments. Also change the envelope delay count on an experimental basis.

Review

To recap this chapter:

- The Color Computer has a built-in digital-to-analog converter (DAC) that can be used in assembly-language routines
- The DAC is operated by outputting 6-bit "left justified" values to location $FF20, a PIA device, in addition to some preliminary routing
- Sound can be generated by outputting a random table of values over repetitive cycles
- Sound can also be generated by outputting a square wave with "envelope" to change the level

For Further Study

"Musical Applications of Microprocessors," Hal Chamberlin, Hayden Book Company, 1980

KEY CHART — CHAPTER 26

INSTRUCTIONS

~~BX~~	~~LBHS~~	~~CLRB~~	~~LDS~~	~~ROL~~
~~DCA~~	~~BITA~~	~~CLR~~	~~LDU~~	~~RORA~~
~~DCB~~	~~BITB~~	~~CMPA~~	~~LDX~~	~~RORB~~
~~DDA~~	~~BLE~~	~~CMPB~~	~~LDY~~	~~ROR~~
~~DDB~~	~~LBLE~~	~~CMPD~~	~~LEAS~~	~~RTI~~
~~DDD~~	~~BLO~~	~~CMPS~~	~~LEAU~~	~~RTS~~
~~NDA~~	~~LBLO~~	~~CMPU~~	~~LEAX~~	~~SBCA~~
~~NDB~~	~~BLS~~	~~CMPX~~	~~LEAY~~	~~SBCB~~
~~NDCC~~	~~LBLS~~	~~CMPY~~	~~LSLA~~	~~SEX~~
~~SLA~~	~~BLT~~	~~COMA~~	~~LSLB~~	~~STA~~
~~SLB~~	~~LBLT~~	~~COMB~~	~~LSL~~	~~STB~~
~~SL~~	~~BMI~~	~~COM~~	~~LSRA~~	~~STD~~
~~SRA~~	~~LBMI~~	~~CWAI~~	~~LSRB~~	~~STS~~
~~SRB~~	~~BNE~~	~~DAA~~	~~LSR~~	~~STU~~
~~SR~~	~~LBNE~~	~~DECA~~	~~MUL~~	~~STX~~
~~CC~~	~~BPL~~	~~DECB~~	~~NEGA~~	~~STY~~
~~BCC~~	~~LBPL~~	~~DEC~~	~~NEGB~~	~~SUBA~~
~~CS~~	~~BRA~~	~~EORA~~	~~NEG~~	~~SUBB~~
~~BCS~~	~~LBRA~~	~~EORB~~	~~NOP~~	~~SUBD~~
~~EQ~~	~~BRN~~	~~EXG~~	~~ORA~~	~~SWI~~
~~BEQ~~	~~LBRN~~	~~INCA~~	~~ORB~~	~~SWI2~~
~~GE~~	~~BSR~~	~~INCB~~	~~ORCC~~	~~SWI3~~
~~BGE~~	~~LBSR~~	~~INC~~	~~PSHS~~	~~SYNC~~
~~GT~~	~~BVC~~	~~JMP~~	~~PSHU~~	~~TFR~~
~~BGT~~	~~LBVC~~	~~JSR~~	~~PULS~~	~~TSTA~~
~~HI~~	~~BVS~~	~~LDA~~	~~PULU~~	~~TSTB~~
~~BHI~~	~~LBVS~~	~~LDB~~	~~ROLA~~	~~TST~~
~~HS~~	~~CLRA~~	~~LDD~~	~~ROLB~~	

EDTASM+ EDITOR COMMANDS

~~A(SSEMBLE)~~	~~I(NSERT)~~	~~R(EPLACE)~~
~~C(OPY)~~	~~L(OAD)~~	~~T(HARDCOPY)~~
~~D(ELETE)~~	~~M(OVE)~~	~~V(ERIFY)~~
~~E(DIT)~~	~~N(UMBER)~~	~~W(RITE)~~
~~F(IND)~~	~~P(RINT)~~	~~Z(BUG)~~
~~H(ARDCOPY)~~	~~Q(UIT)~~	

EDTASM+ ASSEMBLER COMMANDS (A)

~~/AO ABSOLUTE ORIGIN~~	~~/NO NO OBJECT~~
~~/IM IN MEMORY ASSEMBLY~~	~~/NS NO SYMBOL TABLE~~
~~/LP LINE PRINTER~~	~~/SS SHORT SCREEN~~
~~/MO MANUAL ORIGIN~~	~~/WE WAIT ON ERRORS~~
~~/NL NO LISTING~~	

EDTASM+ ZBUG COMMANDS

~~A(SCII) DISPLAY~~	~~T DISPLAY BLOCK~~
~~B(YTE) MODE~~	~~T H HARDCOPY BLOCK~~
~~C(ONTINUE)~~	~~U MOVE BLOCK~~
~~D(ISPLAY)~~	~~V (ERIFY) BLOCK~~
~~E(DITOR)~~	~~W(ORD) MODE~~
~~G(O)~~	~~X BREAKPOINT~~
~~H(ALF) SYMBOLIC~~	~~Y (ANK) BREAKPOINT~~
~~I(NPUT) BASE~~	~~↑ EXAMINE PRECEDING~~
~~L(OAD) ML FILE~~	~~↓ EXAMINE NEXT~~
~~M(NEMONIC) MODE~~	~~→ BRANCH INDIRECT~~
~~N(UMERIC) MODE~~	~~; FORCE NUMERIC~~
~~O(OUTPUT) BASE~~	~~+ FORCE NUMERIC,BYTE~~
~~P SAVE ML ON TAPE~~	~~FORCE FLAGS~~
~~R(EGISTER) DISPLAY~~	~~/ EXAMINE~~
~~S(YMBOLIC DISPLAY)~~	~~, SINGLE STEP~~

ADDRESSING MODES

~~NHERENT~~
~~IRECT~~
~~XTENDED~~
~~MMEDIATE~~
~~IMPLE INDEXED~~
~~ELATIVE~~
~~ISPLACEMENT INDEXED~~
~~UTO INCREMENT/DECREMENT~~
~~NDIRECT~~
~~OPHISTICATED~~

PSEUDO OPS

~~QU~~	~~ORG~~
~~CB~~	~~RMB~~
~~CC~~	~~SET~~
~~DB~~	~~SETDP~~

GENERAL TOPICS

~~CPU REGISTERS~~	~~SUBROUTINES~~
~~DATA TO REGISTERS~~	~~STACK OPERATIONS~~
~~LOADING AND STORING~~	~~ROTATES, SHIFTS~~
~~ADDITION AND SUBTRACTION~~	~~MULTIPLES~~
~~CONDITION CODES~~	~~DIVIDES~~
~~SYMBOLIC ADDRESSING~~	~~DECIMAL ARITHMETIC~~
~~JUMPS, BRANCHES~~	~~BASIC INTERFACING~~
~~RELATIVE BRANCHES~~	~~PASSING PARAMETERS~~
~~INCREMENTS/DECREMENTS~~	~~VARPTR USE~~
~~COMPLEMENTS~~	~~ROM SUBROUTINES~~
~~LOGICAL OPERATIONS~~	~~OTHER ADDRESSING~~
~~MULTIPLE PRECISION~~	~~GRAPHICS~~
~~DATA VALUES~~	~~SOUND~~
~~INDEXING~~	**LARGER PROGRAMS**
~~INDEXING WITH X,Y~~	
~~SORTING~~	

old Type - Present Chapter
egular Type   Future Chapters
~~alic Type~~   *Past Chapters*

Chapter 26
Writing Larger Assembly-Language Programs

In this final chapter we'll give you some tips on how to write larger 6809 programs. The sample programs we've been using here are all short subroutines, but, of course, its possible to write huge assembly-language programs that can accomplish truly wonderful things. The procedure for writing larger programs is fairly standard. We'll show you the steps involved here. The procedure consists of 5 parts — design, flowcharting, coding, debugging, and documentation.

Program Design

This is the first stage of any program, whether it is BASIC programming or assembly-language programming. Program design for large programs in many cases consists of writing the "design specification" for the program before anything else is done.

The design spec is a detailed manual, outlining what the program will do, and in general how the program will go about it. All screen formats, record formats, menus, and commands are listed and detailed.

Of course, for small programs, you don't really need this design spec. If you were implementing a bubble sort, for example, you know that the sort has to put all of the data items of a table in sequence, and not too much more can be said about it.

This design phase of a program is still critical, however, even though you don't write a design spec. You should spend some time thinking about the general "dimensions" of your problem. In the case of a bubble sort, for example, you might ask yourself how many entries will be sorted, what the maximum number of entries are, whether the number of entries can be held in 8 bits or 16, where the table will be located, how large the size of entries will be, and so forth.

If you don't give the problem some thought, you may find yourself in the middle of a program that simply won't work because it can't!

Program Flowcharting

The next step in any type of programming is flowcharting. We've used a few flowcharts in these lessons, so you're somewhat familiar with the symbols.

For a recap, the symbols are shown in Figure 26-1.

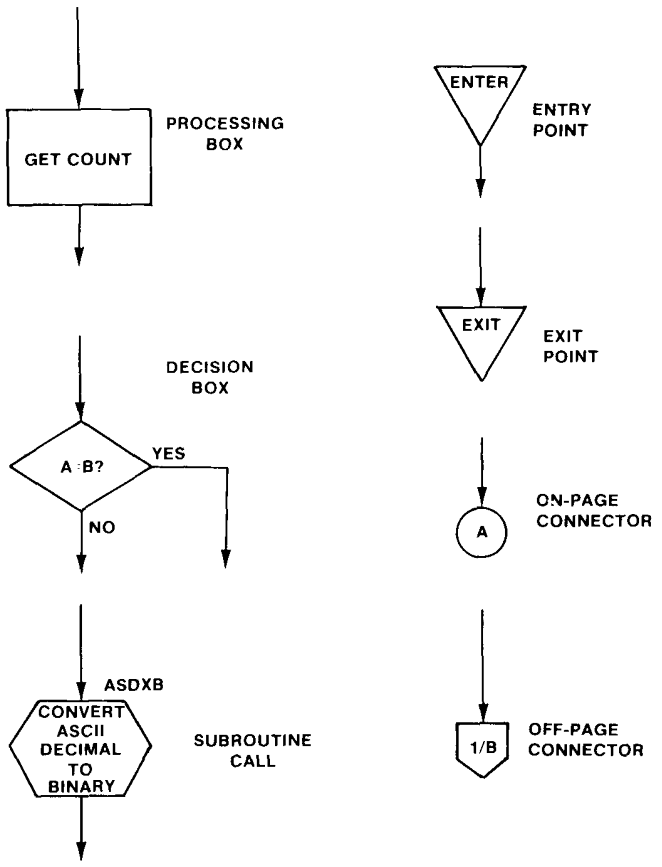


Figure 26-1. Flowcharting Symbols

The rectangular box is a "processing box." Any type of general processing, such as "LOAD NUM WITH 0," "GET NEXT CHARACTER," or "BUMP POINTER BY 1" is put into the box to describe the processing action.

The diamond is a decision box. The decision box would be equated to conditional jumps in assembly language. Two or more exit points might be used from the decision box. You might have something like "A<B?" with one branch labeled "YES" and the other "NO," or you might have "MORE ENTRIES?" with another set of "YES" and "NO" exits.

I've used the six-sided symbol for subroutine processing, although this is not always standard. It's customary to put the name of the subroutine on the upper right of the box. The description within the subroutine box might be something like "SCAN TABLE FOR LEAST ENTRY" or "CONVERT TO BINARY."

The triangles are exit and entry points from a subroutine or other code. Typical descriptive text would be "ENTER" or "ENTER FROM COM-

MAND INTERPRETER" in the entry points and "RETURN" or "RETURN TO MAIN" in the exit points.

The circles are "on-page connectors." They are usually labeled with alphabetical letters. Two circles on each page with have the same designator, showing how the program flows without having to draw confusing lines.

The spade-shaped symbols are "off-page connectors," which connect a program point to a continuing point on another page. The off-page connectors are usually labeled with page numbers.

Typically labels on the left top of each flowchart symbol represent the label that will be used in the assembly listing.

The above flowchart symbols are suggestions only, although the rectangle and diamond are standard symbols.

Flowcharts usually flow down and to the right.

When flowcharts are done before a program is coded, it makes it very easy for a programmer to see what is happening without having to wade through dozens of instructions.

Hints and Kinks 26-1
Flowcharting Template

Radio Shack sells (at this time of writing) a programming flowchart template, a plastic guide with shapes for standard flowcharting symbols. If you do a lot of flowcharting, a template is a decided advantage. If not available at Radio Shack by the time this book appears, the template is commonly found elsewhere.

Do you need to flowchart? Not for simple programs. For programs that involve dozens of instructions, a quick rough flowchart helps clarify what you're going to do, however. For larger programs, flowcharts are a must. They help you plan the code and provide an indication of how you did things when you come back to the program six months later!

Some programmers use notes as an alternative to flowcharting. Use flowcharts first for larger programs, and then find a technique that seems to work for you.

If you are flowcharting your program, try to divide a large program up into subroutines and other "modules," as much as possible, rather than having one huge set of continuous code.

A module would in many cases be a subroutine with the entry conditions and exit conditions very well defined, as we've done in some of the subroutines in this text.

In other cases, a module would simply be a functional processing block that performed a specific function, such as doing the "insert" function of a word-processing program.

Program Coding

After some thought about program design and some flowcharting for larger programs, you're ready to code. You may find that your flowcharts are detailed enough so that you can just sit down and enter instructions directly into the editor/assembler. Chances are, though, you'll have to code your program first using paper and pencil.

Once you've coded the program on paper and given it a cursory check, you're ready to enter it into the editor/assembler.

A few tips about program structure:

- Programs usually flow through from beginning to end.
- Try to utilize as many subroutines and modules as possible.
- Use labels for subroutine and module entry points that correspond to the function — "RDCHAR" for "Read Character," for example. Labels after this entry point may be defined by the first 3 letters of the function and then 3 digits in ascending order. You might have the labels "RDC010," "RDC030," and "RDC100" as three labels in the RDCHAR subroutine, for example.
- Use as many comments as possible.
- Format your programs with "pretty printing." Bracket subroutines with asterisks or other symbols, and try to create listings that are easy to read.
- Use labels that correspond to the flowchart location points. You might want to go back and add the labels to the flowchart after you've coded them.
- Labels are not necessary unless they define jump locations or subroutine locations. They just take up space in the EDTASM+ symbol table otherwise.

When you assemble the program, don't be disappointed if you get dozens of errors. This happens to every programmer at different times. Typically, it might take 3 passes to get rid of all errors.

Once you have a "clean program" without errors, you're ready to do program debugging.

Program Debugging

The first step of debugging is called "desk checking." Sit down at your desk, or on the patio, and carefully go over the program listing. You may even want to "play" computer" and pencil in the registers and stack, and then follow the program flow. Chances are you'll uncover some errors that will necessitate reassembly. Again, don't be dismayed, few programmers can write programs that work the first time.

After you've thoroughly desk checked the code and made any necessary changes and reassemblies, then you're ready for "on-line" debugging.

Hints and Kinks 26-2
Desk Checking

Actually, desk checking is not as important with EDTASM+ as it used to be when I wrote programs in industry. Then it was a fight for "computer time" — 32 programmers on one large, expensive machine. EDTASM+ and the Color Computer are so interactive that you're ready for debugging almost as soon as you make one pass through your flowchart to pick up gross logic errors and have a "clean" assembly-language program. Temper this with your own experiences.

There is no set procedure for debugging, but here are some general guidelines:

- Use breakpoints to narrow down where an error is occurring. Breakpoint at one location and see if the location is reached. If not, go to the halfway point and breakpoint there, and so forth, until you find the spot where the error occurred.
- Always reload the program if it "blows up," instead of restarting ZBUG. Incorrect program operation might have destroyed parts of your program, and this will create further errors or misleading information.
- Be aware of which instructions affect the Condition Codes, and which ones do not. LEAS and LEAU, for example, NEVER affect the Condition Codes.
- Make certain that you handle the stack properly, You should have executed a PSH for every PUL, and an RTS for every BSR or at least reset the stack properly by other instructions.

Program Documentation

Once you have a final version of a program, then sit down and write a brief summary of how it works. Include "internal" tables and data structures, and a description of program variables, if the program is large. If you have a simple subroutine interfacing to BASIC that is commented on the listing, then this step is not necessary.

Review

To review what we've learned in this chapter:

- Program development consists of program design, flowcharting, coding, debugging, and documentation
- The design phase consists of a program specification, or at least some serious thought about what the program is to accomplish and how it will go about it
- Somewhat standardized flowchart symbols are used to layout the program flow in "schematic" form
- Coding should first be done on paper and then entered into EDTASM+
- Debugging consists of desk checking and actual "on-line" debugging using a DEBUG package to trace down errors
- Final documentation may be written for larger programs to describe program operation

For Further Study

EDTASM+ Manual

"More TRS-80 Assembly-Language Programming," Radio Shack 62-2075, by William Barden, Jr. (contains material on program development steps).

Appendix I
6809 Instructions Capsulized

ABX	Adds B to X, result to X. Unsigned.
ADC	Adds immediate or memory operand to A or B plus current state of Carry. Result to A or B.
ADD	Garden variety add of immediate or memory operand to A, B, or D. Result to A, B, or D.
AND	Logical AND of immediate or memory operand to A, B, or CC, with result to A, B, or CC. ANDCC used to reset CC.
ASL	Arithmetic shift left. Actually a logical shift with zero filling lsb. A, B, or memory location may be shifted.
ASR	Arithmetic shift right. Sign extends A, B, or location and shifts one bit right.
BCC	Branch on Carry clear. Branch if C=0. Use LBCC if out of range.
BCS	Branch on Carry set. Branch if C=1. Use LBSC if out of range.
BEQ	Branch if equal. Branch if Z=1. Use LBEQ if out of range.
BGE	Branch if greater or equal. Use for signed comparisons. Use LBGE if out of range.
BGT	Branch if greater than. Use for signed comparisons. Use LBGT if out of range.
BHI	Branch if high. Same as branch if A>B for unsigned numbers. Use LBHI if out of range.
BHS	Branch if high or same. Same as A>=B for unsigned numbers. Use LBHS if out of range.
BIT	Test any bit or bits of A or B by ANDing immediate or memory operand and A or B. CC set on result.
BLE	Branch if less than or equal. Use for signed comparisons. Use LBLE if out of range.
BLO	Branch if lower. Branch if A<B. Use for unsigned comparisons. Use LBLO if out of range.
BLS	Branch if lower or same. Use for unsigned comparisons. Use LBLS if out of range.
BLT	Branch on less than. Use for signed comparisions. Use LBLT if out of range.
BMI	Branch on minus. Branch if result sign is -. Use LBMI if out of range.
BNE	Branch on not equal. Branch if Z=0. Use LBNE if out of range.
BPL	Branch on plus. Branch if result sign is +. Use LBPL if out of range.

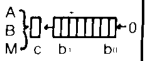
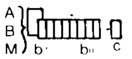
APPENDIX I 6809 Instructions Capsulized

BRA	Branch always. An unconditional relative branch. Use LBRA if out of range.
BRN	Branch Never. A NOP.
BSR	Branch to subroutine. Standard relative call to subroutine. Use LBSR if out of range.
BVC	Branch if no overflow. Use LBVC if out of range.
BVS	Branch if overflow. Use LBVS if out of range.
CLR	Clear the A or B register or a memory location with 0.
CMP	Compare the contents of A, B, D, S, U, X, or Y with an immediate or memory operand. Set all CC except H on result. Standard way of comparing two operands.
COM	One's complement A, B, or memory location. Change all ones to zeroes and all zeroes to ones.
CWAI	Wait for interrupt.
DAA	Decimal adjust A register to change binary result to bcd format. Previous operation must have used bcd operands.
DEC	Decrement 1 from A, B, or memory location. Convenient way of subtracting 1.
EOR	Logical exclusive OR of A or B and immediate or memory operand. Each bit set if one but not both bits in two operands are a one.
EXG	Exchange any two cpu registers of the same length.
INC	Increment 1 from A, B, or memory location. Convenient way of adding 1.
JMP	Absolute (non-relative) unconditional jump to location.
JSR	Jump to subroutine. Another way of calling subroutine.
LD	Load. Standard way of loading A, B, D, S, U, X, or Y with immediate data value or operand from memory.
LEA	Load effective address. 6809 equivalent instruction to increment X or Y. Also used for U and S and with any increment or decrement amount. Can also use one index register in loading another.
LSL	Logical shift left. Shifts A, B, or memory location one bit left, filling lsb with a zero.
LSR	Logical shift right. Shifts A, B, or memory location one bit right, filling msb with a zero.
MUL	Multiplies A by B with result going into D (A contains ms byte, B contains ls byte.) Multiply is unsigned.

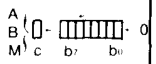
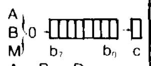
6809 Instructions Capsulized APPENDIX I

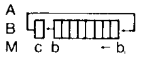
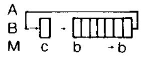
NEG	Two's complement (negate) A, B, or memory location.
NOP	Does nothing. No Condition Codes affected.
OR	Logical OR of immediate or memory operand with A, B, or CC. ORCC is used to set one or more condition codes.
PSH	Push one or more registers on stack. Use the register mnemonics in any order as they are predefined in instruction. U or S stack may be used.
PUL	Pull one or more registers from stack. Use the register mnemonics in any order as they are predefined in instruction. U or S stack may be used.
ROL	Rotate A, B, or memory location one bit left through the Carry Condition code. This is a 9-bit rotate.
ROR	Rotate A, B, or memory location one bit right through the Carry Condition Code. This is a 9-bit rotate.
RTI	Return from interrupt. Interrupt end instruction comparable to RTS.
RTS	Return from subroutine. Pulls return address from S stack and returns to instruction after call.
SBC	Subtract an immediate or memory operand plus any borrow in Carry from contents of A or B. Result is put into A or B.
SEX	Sign extend B into A. If sign bit of B is a one (negative), put $FF into A, otherwise put $00 into A.
ST	Standard store of register to memory. A, B, D, S, U, X, or Y is stored to a memory location or two memory locations.
SUB	Subtract an immediate or memory operand from A, B, or D register. Result stored back into A, B, or D.
SWI	Software interrupt.
SYNC	Synchronize to interrupt.
TFR	Transfer (copy) one register to another. Registers must be of same size.
TST	Test contents of A, B, or memory location by setting N and Z condition Codes based on data in contents.

Appendix II
Detailed Instruction Set

INSTRUCTION/	FORMS	6809 ADDRESSING MODES																		DESCRIPTION	5	3	2	1	0
		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED¹			RELATIVE								
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~⁵	#		H	N	Z	V	C
ABX		3A	3	1																B + X → X (UNSIGNED)	•	•	•	•	•
ADC	ADCA				99	4	2	B9	5	3	89	2	2	A9	4+	2+				A + M + C → A	↕	↕	↕	↕	↕
	ADCB				D9	4	2	F9	5	3	C9	2	2	E9	4+	2+				B + M + C → B	↕	↕	↕	↕	↕
ADD	ADDA				9B	4	2	BB	5	3	8B	2	2	AB	4+	2+				A + M → A	↕	↕	↕	↕	↕
	ADDB				DB	4	2	FB	5	3	CB	2	2	EB	4+	2+				B + M → B	↕	↕	↕	↕	↕
	ADDD				D3	6	2	F3	7	3	C3	4	3	E3	6+	2+				D + M M + 1 → D	↕	↕	↕	↕	↕
AND	ANDA				94	4	2	B4	5	3	84	2	2	A4	4+	2+				A ∧ M → A	•	↕	↕	0	•
	ANDB				D4	4	2	F4	5	3	C4	2	2	E4	4+	2+				B ∧ M → B	•	↕	↕	0	•
	ANDCC										1C	3	2							CC ∧ IMM → CC					1
ASL	ASLA	48	2	1																	8	↕	↕	↕	↕
	ASLB	58	2	1																	8	↕	↕	↕	↕
	ASL				06	6	2	78	7	3				68	6+	2+					8	↕	↕	↕	↕
ASR	ASRA	47	2	1																	8	↕	↕	•	↕
	ASRB	57	2	1																	8	↕	↕	•	↕
	ASR				07	6	2	77	7	3				67	6+	2+					8	↕	↕	•	↕
BCC	BCC																24	3	2	Branch C 0	•	•	•	•	•
	LBCC																10 24	5(6)	4	Long Branch C 0	•	•	•	•	•
BCS	BCS																25	3	2	Branch C 1	•	•	•	•	•
	LBCS																10 25	5(6)	4	Long Branch C = 1	•	•	•	•	•
BEQ	BEQ																27	3	2	Branch Z 0	•	•	•	•	•
	LBEQ																10 27	5(6)	4	Long Branch Z = 0	•	•	•	•	•
BGE	BGE																2C	3	2	Branch ≥ Zero	•	•	•	•	•
	LBGE																10 2C	5(6)	4	Long Branch ≥ Zero	•	•	•	•	•
BGT	BGT																2E	3	2	Branch Zero	•	•	•	•	•
	LBGT																10 2E	5(6)	4	Long Branch > Zero	•	•	•	•	•
BHI	BHI																22	3	2	Branch Higher	•	•	•	•	•
	LBHI																10 22	5(6)	4	Long Branch Higher	•	•	•	•	•
BHS	BHS																24	3	2	Branch Higher or Same	•	•	•	•	•
	LBHS																10 24	5(6)	4	Long Branch Higher or Same	•	•	•	•	•
BIT	BITA				95	4	2	B5	5	3	85	2	2	A5	4+	2+				Bit Test A (M ∧ A)	•	↕	↕	0	•
	BITB				D5	4	2	F5	5	3	C5	2	2	E5	4+	2+				Bit Test B (M ∧ B)	•	↕	↕	0	•
BLE	BLE																2F	3	2	Branch ≤ Zero	•	•	•	•	•
	LBLE																10 2F	5(6)	4	Long Branch ≤ Zero	•	•	•	•	•
BLO	BLO																25	3	2	Branch Lower	•	•	•	•	•
	LBLO																10 25	5(6)	4	Long Branch Lower	•	•	•	•	•
BLS	BLS																23	3	2	Branch Lower or Same	•	•	•	•	•
	LBLS																10 23	5(6)	4	Long Branch Lower or Same	•	•	•	•	•
BLT	BLT																2D	3	2	Branch Zero	•	•	•	•	•
	LBLT																10 2D	5(6)	4	Long Branch < Zero	•	•	•	•	•
BMI	BMI																2B	3	2	Branch Minus	•	•	•	•	•
	LBMI																10 2B	5(6)	4	Long Branch Minus	•	•	•	•	•
BNE	BNE																26	3	2	Branch Z ≠ 0	•	•	•	•	•
	LBNE																10 26	5(6)	4	Long Branch Z ≠ 0	•	•	•	•	•
BPL	BPL																2A	3	2	Branch Plus	•	•	•	•	•
	LBPL																10 2A	5(6)	4	Long Branch Plus	•	•	•	•	•
BRA	BRA																20	3	2	Branch Always	•	•	•	•	•
	LBRA																16	5	3	Long Branch Always	•	•	•	•	•
BRN	BRN																21	3	2	Branch Never	•	•	•	•	•
	LBRN																10 21	5	4	Long Branch Never	•	•	•	•	•

APPENDIX II Detailed Instruction Set

BSR	BSR																8D	7	2	Branch to Subroutine	•	•	•	•	•
	LBSR																17	9	3	Long Branch to Subroutine	•	•	•	•	•
BVC	BVC																28	3	2	Branch V = 0	•	•	•	•	•
	LBVC																10 28	5(6)	4	Long Branch V = 0	•	•	•	•	•
BVS	BVS																29	3	2	Branch V = 1	•	•	•	•	•
	LBVS																10 29	5(6)	4	Long Branch V = 1	•	•	•	•	•
CLR	CLRA	4F	2	1																0 → A	•	0	1	0	0
	CLRB	5F	2	1																0 → B	•	0	1	0	0
	CLR				0F	6	2	7F	7	3				6F	6+	2+				0 → M	•	0	1	0	0
CMP	CMPA				91	4	2	B1	5	3	81	2	2	A1	4+	2+				Compare M from A	8	↕	↕	↕	↕
	CMPB				D1	4	2	F1	5	3	C1	2	2	E1	4+	2+				Compare M from B	8	↕	↕	↕	↕
	CMPD				10 93	7	3	10 B3	8	4	10 83	5	4	10 A3	7+	3+				Compare M:M + 1 from D	•	↕	↕	↕	↕
	CMPS				11 9C	7	3	11 BC	8	4	11 8C	5	4	11 AC	7+	3+				Compare M:M + 1 from S	•	↕	↕	↕	↕
	CMPU				11 93	7	3	11 B3	8	4	11 83	5	4	11 A3	7+	3+				Compare M:M + 1 from U	•	↕	↕	↕	↕
	CMPX				9C	6	2	BC	7	3	8C	4	3	AC	6+	2+				Compare M:M + 1 from X	•	↕	↕	↕	↕
	CMPY				10 9C	7	3	10 BC	8	4	10 8C	5	4	10 AC	7+	3+				Compare M:M + 1 from Y	•	↕	↕	↕	↕
COM	COMA	43	2	1																$\overline{A}$ → A	•	↕	↕	0	1
	COMB	53	2	1																$\overline{B}$ → B	•	↕	↕	0	1
	COM				03	6	2	73	7	3				63	6+	2+				$\overline{M}$ → M	•	↕	↕	0	1
CWAI		3C	20	2																CC ∧ IMM → CC, Wait for Interrupt					1
DAA		19	2	1																Decimal Adjust A	•	↕	↕	0	↕
DEC	DECA	4A	2	1																A − 1 → A	•	↕	↕	↕	•
	DECB	5A	2	1																B − 1 → B	•	↕	↕	↕	•
	DEC				0A	6	2	7A	7	3				6A	6+	2+				M − 1 → M	•	↕	↕	↕	•
EOR	EORA				98	4	2	B8	5	3	88	2	2	A8	4+	2+				A ⊻ M → A	•	↕	↕	0	•
	EORB				D8	4	2	F8	5	3	C8	2	2	E8	4+	2+				B ⊻ M → B	•	↕	↕	0	•
EXG	R1, R2	1E	7	2																R1 ↔ R2[2]	•	•	•	•	•
INC	INCA	4C	2	1																A + 1 → A	•	↕	↕	↕	•
	INCB	5C	2	1																B + 1 → B	•	↕	↕	↕	•
	INC				0C	6	2	7C	7	3				6C	6+	2+				M + 1 → M	•	↕	↕	↕	•
JMP					0E	3	2	7E	4	3				6E	3+	2+				EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	BD	8	3				AD	7+	2+				Jump to Subroutine	•	•	•	•	•
LD	LDA				96	4	2	B6	5	3	86	2	2	A6	4+	2+				M → A	•	↕	↕	0	•
	LDB				D6	4	2	F6	5	3	C6	2	2	E6	4+	2+				M → B	•	↕	↕	0	•
	LDD				DC	5	2	FC	6	3	CC	3	3	EC	5+	2+				M:M + 1 → D	•	↕	↕	0	•
	LDS				10 DE	6	3	10 FE	7	4	10 CE	4	4	10 EE	6+	3+				M:M + 1 → S	•	↕	↕	0	•
	LDU				DE	5	2	FE	6	3	CE	3	3	EE	5+	2+				M:M + 1 → U	•	↕	↕	0	•
	LDX				9E	5	2	BE	6	3	8E	3	3	AE	5+	2+				M:M + 1 → X	•	↕	↕	0	•
	LDY				10 9E	6	3	10 BE	7	4	10 8E	4	4	10 AE	6+	3+				M:M + 1 → Y	•	↕	↕	0	•
LEA	LEAS													32	4+	2+				EA[3] → S	•	•	•	•	•
	LEAU													33	4+	2+				EA[3] → U	•	•	•	•	•
	LEAX													30	4+	2+				EA[3] → X	•	•	↕	•	•
	LEAY													31	4+	2+				EA[3] → Y	•	•	↕	•	•
LSL	LSLA	48	2	1																	•	↕	↕	↕	↕
	LSLB	58	2	1																	•	↕	↕	↕	↕
	LSL				08	6	2	78	7	3				68	6+	2+					•	↕	↕	↕	↕
LSR	LSRA	44	2	1																	•	0	↕	•	↕
	LSRB	54	2	1																	•	0	↕	•	↕
	LSR				04	6	2	74	7	3				64	6+	2+					•	0	↕	•	↕
MUL		3D	11	1																A × B → D (Unsigned)	•	•	↕	•	9
NEG	NEGA	40	2	1																$\overline{A}$ + 1 → A	8	↕	↕	↕	↕
	NEGB	50	2	1																$\overline{B}$ + 1 → B	8	↕	↕	↕	↕
	NEG				00	6	2	70	7	3				60	6+	2+				$\overline{M}$ + 1 → M	8	↕	↕	↕	↕
NOP		12	2	1																No Operation	•	•	•	•	•
OR	ORA				9A	4	2	BA	5	3	8A	2	2	AA	4+	2+				A ∨ M → A	•	↕	↕	0	•
	ORB				DA	4	2	FA	5	3	CA	2	2	EA	4+	2+				B ∨ M → B	•	↕	↕	0	•
	ORCC										1A	3	2							CC ∨ IMM → CC					7
PSH	PSHS	34	$5+^{4}$	2																Push Registers on S Stack	•	•	•	•	•
	PSHU	36	$5+^{4}$	2																Push Registers on U Stack	•	•	•	•	•

PUL	PULS	35	5+[1]	2															Pull Registers from S Stack	•	•	•	•	•
	PULU	37	5+[1]	2															Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA	49	2	1															A	•	↕	↕	↕	↕
	ROLB	59	2	1															B 	•	↕	↕	↕	↕
	ROL				09	6	2	79	7	3				69	6+	2+				M c b ← b	•	↕	↕	↕	↕
ROR	RORA	46	2	1															A	•	↕	↕	•	↕
	RORB	56	2	1															B 	•	↕	↕	•	↕
	ROR				06	6	2	76	7	3				66	6+	2+				M c b → b	•	↕	↕	•	↕
RTI		3B	6/15	1															Return From Interrupt					7
RTS		39	5	1															Return From Subroutine	•	•	•	•	•
SBC	SBCA				92	4	2	B2	5	3	82	2	2	A2	4+	2+				A − M − C → A	8	↕	↕	↕	↕
	SBCB				D2	4	2	F2	5	3	C2	2	2	E2	4+	2+				B − M − C → B	8	↕	↕	↕	↕
SEX		1D	2	1															Sign Extend B into A	•	↕	↕	0	•
ST	STA				97	4	2	B7	5	3				A7	4+	2+				A → M	•	↕	↕	0	•
	STB				D7	4	2	F7	5	3				E7	4+	2+				B → M	•	↕	↕	0	•
	STD				DD	5	2	FD	6	3				ED	5+	2+				D → M:M + 1	•	↕	↕	0	•
	STS				10 DF	6	3	10 FF	7	4				10 EF	6+	3+				S → M:M + 1	•	↕	↕	0	•
	STU				DF	5	2	FF	6	3				EF	5+	2+				U → M:M + 1	•	↕	↕	0	•
	STX				9F	5	2	BF	6	3				AF	5+	2+				X → M:M + 1	•	↕	↕	0	•
	STY				10 9F	6	3	10 BF	7	4				10 AF	6+	3+				Y → M:M + 1	•	↕	↕	0	•
SUB	SUBA				90	4	2	B0	5	3	80	2	2	A0	4+	2+				A − M → A	8	↕	↕	↕	↕
	SUBB				D0	4	2	F0	5	3	C0	2	2	E0	4+	2+				B − M → B	8	↕	↕	↕	↕
	SUBD				93	6	2	B3	7	3	83	4	3	A3	6+	2+				D − M:M + 1 → D	•	↕	↕	↕	↕
SWI	SWI[6]	3F	19	1															Software Interrupt 1	•	•	•	•	•
	SWI2[4]	10 3F	20	2															Software Interrupt 2	•	•	•	•	•
	SWI3[4]	11 3F	20	2															Software Interrupt 3	•	•	•	•	•
SYNC		13	≥2	1															Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2	1F	7	2															R1 → R2[1]	•	•	•	•	•
TST	TSTA	4D	2	1															Test A	•	↕	↕	0	•
	TSTB	5D	2	1															Test B	•	↕	↕	0	•
	TST				0D	6	2	7D	7	3				6D	6+	2+				Test M	•	↕	↕	0	•

INDEXED ADDRESSING MODES

		NON INDIRECT				INDIRECT			
TYPE	FORMS	Assembler Form	Post-Byte OP Code	+ ~	+ #	Assembler Form	Post-Byte OP Code	+ ~	+ #
CONSTANT OFFSET FROM R	NO OFFSET	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5 BIT OFFSET	n, R	0RRnnnnn	1	0	defaults to 8-bit			
	8 BIT OFFSET	n, R	1RR01000	1	1	[n, R]	1RR11000	4	1
	16 BIT OFFSET	n, R	1RR01001	4	2	[n, R]	1RR11001	7	2
ACCUMULATOR OFFSET FROM R	A—REGISTER OFFSET	A, R	1RR00110	1	0	[A, R]	1RR10110	4	0
	B—REGISTER OFFSET	B, R	1RR00101	1	0	[B, R]	1RR10101	4	0
	D—REGISTER OFFSET	D, R	1RR01011	4	0	[D, R]	1RR11011	7	0
AUTO INCREMENT/DECREMENT R	INCREMENT BY 1	,R+	1RR00000	2	0	not allowed			
	INCREMENT BY 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	DECREMENT BY 1	,-R	1RR00010	2	0	now allowed			
	DECREMENT BY 2	,--R	1RR00011	3	0	[,--R]	1RR10001	6	0
CONSTANT OFFSET FROM PC	8 BIT OFFSET	n, PCR	1XX01100	1	1	[n, PCR]	1XX11100	4	1
	16 BIT OFFSET	n, PCR	1XX01101	5	2	[n, PCR]	1XX11101	8	2
EXTENDED INDIRECT	16 BIT ADDRESS	—	—	—	—	[n]	10011111	5	2
	R = X, Y, U, or S X = DON'T CARE								

APPENDIX II Detailed Instruction Set

NOTES:

1. Given in the table are the base cycles and byte counts. To determine the total cycles and byte counts add the values from the '6809 indexing modes' table.
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.
   The 8 bit registers are: A, B, CC, DP
   The 16 bit registers are: X, Y, U, S, D, PC
3. EA is the effective address.
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each *byte* pushed or pulled.
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken.
6. SW1 sets I&F bits. SW12 and SW13 do not affect I&F.
7. Conditions Codes set as a direct result of the instruction.
8. Value of half-carry flag is undefined.
9. Special Case—Carry set if b7 is SET.

LEGEND:

OP	Operation Code (Hexadecimal);	Z	Zero (byte)
~	Number of MPU Cycles;	V	Overflow, 2's complement
#	Number of Program Bytes;	C	Carry from bit 7
+	Arithmetic Plus;	↑	Test and set if true, cleared otherwise
-	Arithmetic Minus;	●	Not Affected
*	Multiply	CC	Condition Code Register
$\overline{M}$	Complement of M;	:	Concatenation
→	Transfer into;	∨	Logical or
H	Half-carry from bit 3;	∧	Logical and
N	Negative (sign bit)	⊻	Logical Exclusive or

Appendix III
EDTASM+ Commands

Editor Commands

A	Assemble	Assembles current text. See Assembler Commands.
C	Copy	Copies a block of lines to a new position. Source block remains unchanged. C1000,100:130,5 copies 100-130 to new set of lines starting with 1000 with increment 5.
D	Delete	Deletes one or more lines. D deletes current line, D100 deletes line 100, D300:500 deletes lines 300-500.
E	Edit	Enter Edit subcommand mode. E100 enters Edit subcommand mode for line 100.
F	Find	Find one or more characters in text buffer. FA searches for "A," FLABEL searches for "LABEL." F without string searches for last string.
H	Hardcopy	List a set of lines to system printer. H prints current line, H100 prints line 100, H#:* prints entire buffer, H100:300 prints lines 100-300.
I	Insert	Enter insert mode. I301,1 inserts from line 301 with increment 1. I301 inserts from 301 with last increment.
L	Load	Loads cassette source file. L loads next file from cassette. L NAME loads file NAME.
M	Move	Moves a block of lines from one position to another. M1000,100:300,10 moves lines 100 through 300 to new position starting with line number 1000 and increment 10.
N	Renumber	Renumbers text lines. N100,10 renumbers starting with line 100 and increment 10.
P	Display	Displays text lines on screen. P# displays first line, P100:300 displays lines 100-300, P#:* displays entire buffer.
Q	Quit	Returns to BASIC.
R	Replace	Replaces one line and then enters insert mode. R100,10 replaces line 100 and then enters insert mode with line increment 10.
T	Print	Like H except line **numbers** are not printed.
V	Verify	Verifies previously written cassette file with contents of text buffer. V verifies next file, V NAME verifies file NAME.

W	Write	Writes text buffer to cassette. W writes text as file NONAME, W NAME writes file as NAME.
Z	ZBUG	Transfers control to ZBUG.

A Commands for Assembly

/AO	Assemble with absolute origin
/IM	Assemble in memory
/LP	Listing to Line Printer
/MO	Use manual origin
/NL	No listing
/NO	No object
/NS	No symbol table
/WE	Wait on errors

ZBUG Commands

A	ASCII	Display in ASCII. Non-ASCII as blanks.
B	Byte mode	Set Byte mode. Display contents as 8-bit value.
C	Continue	Continue from breakpoint.
D	Display	Display current breakpoints.
E	Editor	Return to Editor.
G	Go	Execute from location. GSTART starts from location START, GA00 starts from $A00.
H	Half symbolic	Display instruction addresses as numeric, but instructions as mnemonic.
I	Input base	Set input base. I10 sets decimal.
L	Load	Load machine-language file. L loads next cassette file, L NAME loads file NAME.
M	Mnemonic mode	Set mnemonic mode. Instructions displayed in their mnemonic form.
N	Numeric mode	Set numeric mode. Data displayed as numeric values rather than instruction mnemonics.
O	Output base	Set output base. O16 sets hexadecimal.
P	Save tape	Dump memory to cassette as machine-language or memory image. PNAME 3000 3010 3000 dumps locations 3000 through 3010 as file NAME with execution address 3000. Use dummy execution address for data.

R	Display registers	Displays contents of all registers.
S	Symbolic mode	Display addresses in symbolic form based upon current contents of symbol table.
T	Display block	Display a block of locations. T3000 3010 displays locations 3000 through 3010.
TH	Hardcopy block	Print a block of locations. Similar to T.
U	Move block	Move a block of locations from one memory location to another. M3000 4000 10 moves the 16 locations from 3000 through 300F to locations 4000 through 400F.
V	Verify	Verifies machine language file. V verifies next cassette file, V NAME verifies file NAME.
W	Word mode	Sets word mode. Data is displayed as 16-bit words.
X	Breakpoint	Set breakpoint. XLOOP sets breakpoint at location LOOP, XA00 sets breakpoint at location A00.
Y	Yank break-point	Kills breakpoint. Y kills all breakpoints, Y3 kills breakpoint number 3.
↑	Examine preceding	Examine preceding location.
↓	Examine next	Examine next location, increment based on current modes.
→	Branch indirect	Treat current display data as address, and reset display location to that address.
;	Force numeric	One time display of current data to numeric.
=	Force numeric byte	One time display of current data to numeric and byte modes.
:	Force flags	Convert current data to flag mnemonics.
/	Examine	Examine (display) register or memory location. A/ examines A register, LABEL/ examines location LABEL, A00/ examines location $A00.
,	Single step	Execute one instruction. START, starts at START and executes one instruction, comma alone continues from current instruction location.

Appendix IV
Binary/Decimal/Hexadecimal Conversions

HEX	BINARY	DECIMAL
00	00000000	0
01	00000001	1
02	00000010	2
03	00000011	3
04	00000100	4
05	00000101	5
06	00000110	6
07	00000111	7
08	00001000	8
09	00001001	9
0A	00001010	10
0B	00001011	11
0C	00001100	12
0D	00001101	13
0E	00001110	14
0F	00001111	15
10	00010000	16
11	00010001	17
12	00010010	18
13	00010011	19
14	00010100	20
15	00010101	21
16	00010110	22
17	00010111	23
18	00011000	24
19	00011001	25
1A	00011010	26
1B	00011011	27
1C	00011100	28
1D	00011101	29
1E	00011110	30
1F	00011111	31
20	00100000	32
21	00100001	33
22	00100010	34
23	00100011	35
24	00100100	36
25	00100101	37
26	00100110	38
27	00100111	39
28	00101000	40
29	00101001	41
2A	00101010	42
2B	00101011	43
2C	00101100	44
2D	00101101	45
2E	00101110	46
2F	00101111	47
30	00110000	48
31	00110001	49
32	00110010	50
33	00110011	51
34	00110100	52
35	00110101	53
36	00110110	54
37	00110111	55
38	00111000	56
39	00111001	57
3A	00111010	58
3B	00111011	59
3C	00111100	60
3D	00111101	61
3E	00111110	62

HEX	BINARY	DECIMAL
3F	00111111	63
40	01000000	64
41	01000001	65
42	01000010	66
43	01000011	67
44	01000100	68
45	01000101	69
46	01000110	70
47	01000111	71
48	01001000	72
49	01001001	73
4A	01001010	74
4B	01001011	75
4C	01001100	76
4D	01001101	77
4E	01001110	78
4F	01001111	79
50	01010000	80
51	01010001	81
52	01010010	82
53	01010011	83
54	01010100	84
55	01010101	85
56	01010110	86
57	01010111	87
58	01011000	88
59	01011001	89
5A	01011010	90
5B	01011011	91
5C	01011100	92
5D	01011101	93
5E	01011110	94
5F	01011111	95
60	01100000	96
61	01100001	97
62	01100010	98
63	01100011	99
64	01100100	100
65	01100101	101
66	01100110	102
67	01100111	103
68	01101000	104
69	01101001	105
6A	01101010	106
6B	01101011	107
6C	01101100	108
6D	01101101	109
6E	01101110	110
6F	01101111	111
70	01110000	112
71	01110001	113
72	01110010	114
73	01110011	115
74	01110100	116
75	01110101	117
76	01110110	118
77	01110111	119
78	01111000	120
79	01111001	121
7A	01111010	122
7B	01111011	123
7C	01111100	124
7D	01111101	125
7E	01111110	126
7F	01111111	127
80	10000000	128
81	10000001	129
82	10000010	130
83	10000011	131

HEX	BINARY	DECIMAL
84	10000100	132
85	10000101	133
86	10000110	134
87	10000111	135
88	10001000	136
89	10001001	137
8A	10001010	138
8B	10001011	139
8C	10001100	140
8D	10001101	141
8E	10001110	142
8F	10001111	143
90	10010000	144
91	10010001	145
92	10010010	146
93	10010011	147
94	10010100	148
95	10010101	149
96	10010110	150
97	10010111	151
98	10011000	152
99	10011001	153
9A	10011010	154
9B	10011011	155
9C	10011100	156
9D	10011101	157
9E	10011110	158
9F	10011111	159
A0	10100000	160
A1	10100001	161
A2	10100010	162
A3	10100011	163
A4	10100100	164
A5	10100101	165
A6	10100110	166
A7	10100111	167
A8	10101000	168
A9	10101001	169
AA	10101010	170
AB	10101011	171
AC	10101100	172
AD	10101101	173
AE	10101110	174
AF	10101111	175
B0	10110000	176
B1	10110001	177
B2	10110010	178
B3	10110011	179
B4	10110100	180
B5	10110101	181
B6	10110110	182
B7	10110111	183
B8	10111000	184
B9	10111001	185
BA	10111010	186
BB	10111011	187
BC	10111100	188
BD	10111101	189
BE	10111110	190
BF	10111111	191
C0	11000000	192
C1	11000001	193
C2	11000010	194
C3	11000011	195
C4	11000100	196
C5	11000101	197
C6	11000110	198
C7	11000111	199
C8	11001000	200

HEX	BINARY	DECIMAL
C9	11001001	201
CA	11001010	202
CB	11001011	203
CC	11001100	204
CD	11001101	205
CE	11001110	206
CF	11001111	207
D0	11010000	208
D1	11010001	209
D2	11010010	210
D3	11010011	211
D4	11010100	212
D5	11010101	213
D6	11010110	214
D7	11010111	215
D8	11011000	216
D9	11011001	217
DA	11011010	218
DB	11011011	219
DC	11011100	220
DD	11011101	221
DE	11011110	222
DF	11011111	223
E0	11100000	224
E1	11100001	225
E2	11100010	226
E3	11100011	227
E4	11100100	228
E5	11100101	229
E6	11100110	230
E7	11100111	231
E8	11101000	232
E9	11101001	233
EA	11101010	234
EB	11101011	235
EC	11101100	236
ED	11101101	237
EE	11101110	238
EF	11101111	239
F0	11110000	240
F1	11110001	241
F2	11110010	242
F3	11110011	243
F4	11110100	244
F5	11110101	245
F6	11110110	246
F7	11110111	247
F8	11111000	248
F9	11111001	249
FA	11111010	250
FB	11111011	251
FC	11111100	252
FD	11111101	253
FE	11111110	254
FF	11111111	255

To Convert From Binary or Hexadecimal to Decimal:

1. If number is 8 bits or less, use table.
2. If number is greater than 8 bits, use this method:
   A. Divide into two bytes (add zeroes to left if necessary)
   B. Convert first (most significant) by table.
   C. Multiply decimal equivalent of first by 256.
   D. Convert second (least significant) by table.
   E. Add the results of C and D together to find the decimal number.
   F. Example: Convert $AA88 to decimal.
      a. First byte is AA, second is 88.
      b. From table, first is 170 in decimal.
      c. The value 170 multiplied by 256 is 43,520.
      d. Second byte of 88 is 136 decimal from table.
      e. 43,520+136 is 43,656 decimal = $AA88.

To Convert From Decimal to Binary or Hexadecimal:

1. If number is 255 or less, use table.
2. If number is greater than 255, use this method:
   A. Divide by 256 to get an integer result and a remainder.
   B. Convert integer result to a hexadecimal or binary number by table.
   C. Convert remainder to a hexadecimal or binary number by table.
   D. Write down the hex number from B followed by hex number from C; you should have four hex digits or 16 binary digits. The result is the number in hex or binary.
   E. Example: convert 60000 to hexadecimal.
      a. 60000/256=234, remainder of 96.
      b. From table, integer 234 is EA in hexadecimal.
      c. From table, remainder 96 is 60 in hexadecimal.
      d. EA followed by 60 is $EA60 = 60000 decimal.

Appendix V
Two's Complement Numbers

HEX	BINARY	DECIMAL
7F	01111111	+ 127
7E	01111110	+ 126
7D	01111101	+ 125
7C	01111100	+ 124
7B	01111011	+ 123
7A	01111010	+ 122
79	01111001	+ 121
78	01111000	+ 120
77	01110111	+ 119
76	01110110	+ 118
75	01110101	+ 117
74	01110100	+ 116
73	01110011	+ 115
72	01110010	+ 114
71	01110001	+ 113
70	01110000	+ 112
6F	01101111	+ 111
6E	01101110	+ 110
6D	01101101	+ 109
6C	01101100	+ 108
6B	01101011	+ 107
6A	01101010	+ 106
69	01101001	+ 105
68	01101000	+ 104
67	01100111	+ 103
66	01100110	+ 102
65	01100101	+ 101
64	01100100	+ 100
63	01100011	+ 99
62	01100010	+ 98
61	01100001	+ 97
60	01100000	+ 96
5F	01011111	+ 95
5E	01011110	+ 94
5D	01011101	+ 93
5C	01011100	+ 92
5B	01011011	+ 91
5A	01011010	+ 90
59	01011001	+ 89
58	01011000	+ 88
57	01010111	+ 87
56	01010110	+ 86
55	01010101	+ 85
54	01010100	+ 84
53	01010011	+ 83
52	01010010	+ 82
51	01010001	+ 81
50	01010000	+ 80
4F	01001111	+ 79
4E	01001110	+ 78
4D	01001101	+ 77
4C	01001100	+ 76
4B	01001011	+ 75
4A	01001010	+ 74
49	01001001	+ 73
48	01001000	+ 72
47	01000111	+ 71
46	01000110	+ 70
45	01000101	+ 69
44	01000100	+ 68
43	01000011	+ 67
42	01000010	+ 66
41	01000001	+ 65
40	01000000	+ 64
3F	00111111	+ 63
3E	00111110	+ 62

APPENDIX V Two's Complement Numbers

HEX	BINARY	DECIMAL
3D	ØØ111101	+ 61
3C	ØØ1111ØØ	+ 6Ø
3B	ØØ111Ø11	+ 59
3A	ØØ111Ø1Ø	+ 58
39	ØØ111ØØ1	+ 57
38	ØØ111ØØØ	+ 56
37	ØØ11Ø111	+ 55
36	ØØ11Ø11Ø	+ 54
35	ØØ11Ø1Ø1	+ 53
34	ØØ11Ø1ØØ	+ 52
33	ØØ11ØØ11	+ 51
32	ØØ11ØØ1Ø	+ 5Ø
31	ØØ11ØØØ1	+ 49
3Ø	ØØ11ØØØØ	+ 48
2F	ØØ1Ø1111	+ 47
2E	ØØ1Ø111Ø	+ 46
2D	ØØ1Ø11Ø1	+ 45
2C	ØØ1Ø11ØØ	+ 44
2B	ØØ1Ø1Ø11	+ 43
2A	ØØ1Ø1Ø1Ø	+ 42
29	ØØ1Ø1ØØ1	+ 41
28	ØØ1Ø1ØØØ	+ 4Ø
27	ØØ1ØØ111	+ 39
26	ØØ1ØØ11Ø	+ 38
25	ØØ1ØØ1Ø1	+ 37
24	ØØ1ØØ1ØØ	+ 36
23	ØØ1ØØØ11	+ 35
22	ØØ1ØØØ1Ø	+ 34
21	ØØ1ØØØØ1	+ 33
2Ø	ØØ1ØØØØØ	+ 32
1F	ØØØ11111	+ 31
1E	ØØØ1111Ø	+ 3Ø
1D	ØØØ111Ø1	+ 29
1C	ØØØ111ØØ	+ 28
1B	ØØØ11Ø11	+ 27
1A	ØØØ11Ø1Ø	+ 26
19	ØØØ11ØØ1	+ 25
18	ØØØ11ØØØ	+ 24
17	ØØØ1Ø111	+ 23
16	ØØØ1Ø11Ø	+ 22
15	ØØØ1Ø1Ø1	+ 21
14	ØØØ1Ø1ØØ	+ 2Ø
13	ØØØ1ØØ11	+ 19
12	ØØØ1ØØ1Ø	+ 18
11	ØØØ1ØØØ1	+ 17
1Ø	ØØØ1ØØØØ	+ 16
ØF	ØØØØ1111	+ 15
ØE	ØØØØ111Ø	+ 14
ØD	ØØØØ11Ø1	+ 13
ØC	ØØØØ11ØØ	+ 12
ØB	ØØØØ1Ø11	+ 11
ØA	ØØØØ1Ø1Ø	+ 1Ø
Ø9	ØØØØ1ØØ1	+ 9
Ø8	ØØØØ1ØØØ	+ 8
Ø7	ØØØØØ111	+ 7
Ø6	ØØØØØ11Ø	+ 6
Ø5	ØØØØØ1Ø1	+ 5
Ø4	ØØØØØ1ØØ	+ 4
Ø3	ØØØØØØ11	+ 3
Ø2	ØØØØØØ1Ø	+ 2
Ø1	ØØØØØØØ1	+ 1
ØØ	ØØØØØØØØ	+ Ø
FF	11111111	− 1
FE	1111111Ø	− 2
FD	111111Ø1	− 3
FC	111111ØØ	− 4
FB	11111Ø11	− 5
FA	11111Ø1Ø	− 6
F9	11111ØØ1	− 7

HEX	BINARY	DECIMAL
F8	11111000	- 8
F7	11110111	- 9
F6	11110110	- 10
F5	11110101	- 11
F4	11110100	- 12
F3	11110011	- 13
F2	11110010	- 14
F1	11110001	- 15
F0	11110000	- 16
EF	11101111	- 17
EE	11101110	- 18
ED	11101101	- 19
EC	11101100	- 20
EB	11101011	- 21
EA	11101010	- 22
E9	11101001	- 23
E8	11101000	- 24
E7	11100111	- 25
E6	11100110	- 26
E5	11100101	- 27
E4	11100100	- 28
E3	11100011	- 29
E2	11100010	- 30
E1	11100001	- 31
E0	11100000	- 32
DF	11011111	- 33
DE	11011110	- 34
DD	11011101	- 35
DC	11011100	- 36
DB	11011011	- 37
DA	11011010	- 38
D9	11011001	- 39
D8	11011000	- 40
D7	11010111	- 41
D6	11010110	- 42
D5	11010101	- 43
D4	11010100	- 44
D3	11010011	- 45
D2	11010010	- 46
D1	11010001	- 47
D0	11010000	- 48
CF	11001111	- 49
CE	11001110	- 50
CD	11001101	- 51
CC	11001100	- 52
CB	11001011	- 53
CA	11001010	- 54
C9	11001001	- 55
C8	11001000	- 56
C7	11000111	- 57
C6	11000110	- 58
C5	11000101	- 59
C4	11000100	- 60
C3	11000011	- 61
C2	11000010	62
C1	11000001	- 63
C0	11000000	- 64
BF	10111111	- 65
BE	10111110	- 66
BD	10111101	- 67
BC	10111100	- 68
BB	10111011	- 69
BA	10111010	- 70
B9	10111001	- 71
B8	10111000	- 72
B7	10110111	- 73
B6	10110110	- 74
B5	10110101	- 75
B4	10110100	- 76
B3	10110011	- 77

APPENDIX V Two's Complement Numbers

HEX	BINARY	DECIMAL
B2	10110010	-78
B1	10110001	-79
B0	10110000	-80
AF	10101111	-81
AE	10101110	-82
AD	10101101	-83
AC	10101100	-84
AB	10101011	-85
AA	10101010	-86
A9	10101001	-87
A8	10101000	-88
A7	10100111	-89
A6	10100110	-90
A5	10100101	-91
A4	10100100	-92
A3	10100011	-93
A2	10100010	-94
A1	10100001	-95
A0	10100000	-96
9F	10011111	-97
9E	10011110	-98
9D	10011101	-99
9C	10011100	-100
9B	10011011	-101
9A	10011010	-102
99	10011001	-103
98	10011000	-104
97	10010111	-105
96	10010110	-106
95	10010101	-107
94	10010100	-108
93	10010011	-109
92	10010010	-110
91	10010001	-111
90	10010000	-112
8F	10001111	-113
8E	10001110	-114
8D	10001101	-115
8C	10001100	-116
8B	10001011	-117
8A	10001010	-118
89	10001001	-119
88	10001000	-120
87	10000111	-121
86	10000110	-122
85	10000101	-123
84	10000100	-124
83	10000011	-125
82	10000010	-126
81	10000001	-127
80	10000000	-128

Appendix VI
ASCII Codes for the Color Computer (EDTASM+ Generated)

HEX	DECIMAL	ASCII
20	32	(space)
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H

49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	left bracket
5C	92	reverse slash
5D	93	right bracket
5E	94	up arrow
5F	95	left arrow

Index

Radio Shack
A Division of Tandy Corporation
Fort Worth, Texas 76102