CHAPTER 6

PROGRAMMING THE SYM-1

Creating a program on the SYM-1 involves several steps. First, the input to the program and its desired output must be carefully defined. The flow of program logic is usually worked out graphically in the form of a flowchart. Next, the symbols on the flowchart are converted to assembly language instructions. These instructions are in turn translated into machine language, which is entered into memory and executed. If (as usual) the program does not run correctly the first time, you must debug it to uncover the errors in the program. This chapter illustrates the steps involved in creating a program to add two 16-bit binary numbers, and provides two other programming problems with suggested solutions. All three programs are designed to communicate basic programming principles and techniques and to demonstrate a programmer's approach to simple problems.

6.1 HARDWARE

All the sample programs listed here can be loaded and run on the basic SYM-1 with the minimum RAM. The only I/O devices required are the on-board keyboard and display.

If a printing or display terminal is available, by all means use it instead of the Hex keyboard provided. Both types are more comfortable for most users and allow much more data to be displayed at once.

Connect the terminal cable to the appropriate connector on the left edge of the card as described in Chapter 3. Verify that the switches on the terminal are set for full-duplex operation and no parity. The duplexing mode switch will usually be labelled HALF/FULL or H/F; the parity switch will be labelled EVEN/ODD/NO. If your terminal has a CRT, wait for it to warm up. To log on to a terminal, enter a "Q" immediately after reset.

6.2 DOUBLE-PRECISION ADDITION

Since the eight bits of the accumulator can represent positive values only in the range 0-255 (00-FF Hex), 255 is the largest sum that can be obtained by simply loading one 8-bit number into the accumulator and adding another. But by utilizing the Carry Flag, which is set to "1" whenever the result of an addition exceeds 255, multiple-byte numbers may be added and the results stored in memory. A 16-bit sum can represent values greater than 65,000 (up to FFFF Hex). Adding 16-bit rather than 8-bit numbers is called "double-precision" addition, using 24-bit numbers yields triple precision, etc.

6.2.1 Defining Program Flow

Flowcharting is an orderly way of representing a procedure. Much easier to follow than a list of instructions, a flowchart facilitates debugging and also serves as a handy reference when using a program written weeks or months earlier. Some common flowcharting symbols are shown in Figure 6-1. below.

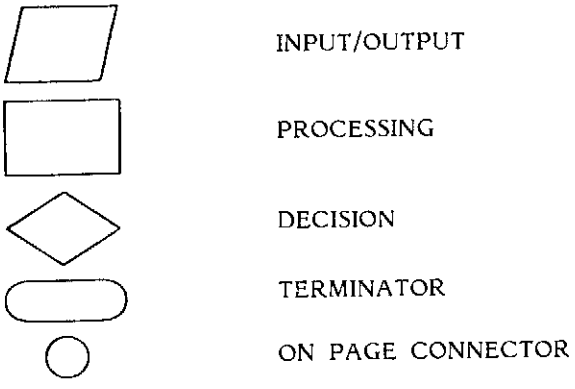


Figure 6-1. COMMON FLOWCHARTING SYMBOLS

The object of our program is to add two 16-bit numbers, each stored in two bytes of RAM, and obtain a 16-bit result. The sequence of operations the processor must perform is shown in the flowchart in Figure 6-2.

To accomplish double-precision addition, first clear the Decimal Mode and the Carry Flags. (The addition is in binary, so the system must not be expecting decimal numbers. The Carry Flag is used in the program and must start at zero.) Load the low byte of the first 16-bit number into the accumulator and add the low order byte of the second number using an Add With Carry (ADC) command. The contents of the accumulator are the low order byte of the result. The Carry Flag is set if the low-byte sum was greater than FF (Hex).

You now store the accumulator contents in memory, load the high order byte of the first number into the accumulator, and add the high order byte of the second number. The ADC command automatically adds the carry bit if it is set. After the second addition, the contents of the accumulator are the high order byte of the result. The example below shows the addition of 384 and 128.

```
0000  0001  1000  0000     384  (0180 Hex)
0000  0000  1000  0000     128  (0080 Hex)

Add low order bytes:  (clear carry)

          1000  0000
          1000  0000
Carry = 1 ----------
          0000  0000

Add high order bytes:  (carry = 1)

          0000  0001
          0000  0000
                   1  CARRY
Carry = 0 ----------
          0000  0010

Result = 0000 0010  0000  0000 = 512  (0200 Hex)
```

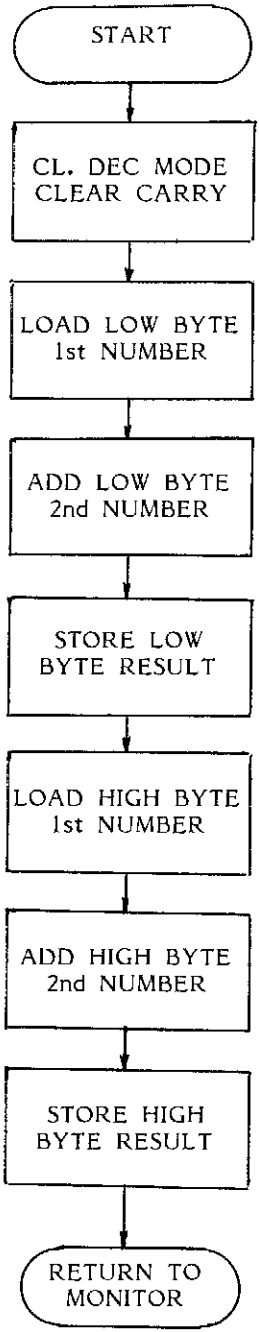


Figure 6-2. DOUBLE-PRECISION ADDITION FLOWCHART

6.2.2 Coding and "Hand Assembly"

Once you have flowcharted a program, you may "code" it onto a form like the one shown in Figure 6-3. SY6502 Microprocessor Assembly Language is described in Sections 4.3.3 and 4.3.4. Additional information is available in the Synertek "Programming Manual" (MNA-2) for the 6500 family. Figure 6-4 shows the coding for our example.

The first step involves finding the SY6502 commands that correspond to the operations specified in the flowchart. A summary of the commands and their mnemonic codes is given in Table 4-7. Arbitrary labels were assigned to represent the addresses of the monitor, the two addends and the sum and entered in the operand field. As written, the assembly language program does not specify where in memory the program and data will be stored.

To store and execute the program, you must "assemble" it by translating the mnemonics into hexadecimal command codes and assign the program to a set of addresses in user RAM. Performing this procedure with pencil and paper, rather than with a special assembler program, is "hand assembly".

The SUPERMON monitor begins at Hex location 8000, and the addends and the sum have been arbitrarily assigned to locations 0301 through 0306. You should note that the high and low order bytes of a 16-bit number need not be stored in contiguous locations, although they are in this example.

The program will be stored beginning in location 0200, another arbitrary choice. Data and programs may be stored anywhere in user RAM. Columns B1, B2, and B3 represent the three possible bytes in any 6502 instruction. B1 always contains the Hexadecimal operation code. B2 and B3 represent the operand(s). Looking at the coding form, you can see that the CLD and CLC instructions each occupy one byte and that the LDA instruction occupies three bytes. On your instruction set summary card, you'll see that the LDA mnemonic represents several different operation codes depending on the addressing mode chosen. AD indicates absolute addressing and specifies a three-byte command. When all the operation codes and operands have been translated into pairs of Hex digits, the program is ready to be entered into memory and executed.

6.2.3 Entering and Executing the Program

The procedure for entering the double precision addition program is shown below.

YOU KEY IN	DISPLAY SHOWS	EXPLANATION
(RST)		
(CR)	SY1.1..	
(MEM) 200 (CR)	0200.**.	Enter memory display and modify mode
D8	0201.**.	Store D8 in location 0200, advance to next location
18	0202.**.	Store 18 in location 0201, advance to next location
AD	0203.**.	.
02	0204.**.	.
03	0205.**.	.
6D	0206.**.	.
	.	
	.	
	.	
80	0217.**.	
(CR)	217.**..	Exit memory display and modify mode

SYNERTEK SYSTEMS CORPORATION

PROGRAM _____

PROGRAMMER _____

DATE _____

ADDR	INSTRUCTIONS B1	B2	B3	LABEL	MNEMONIC	OPERAND	COMMENTS

Figure 6-3. SAMPLE CODING FORM

SYNERTEK SYSTEMS CORPORATION

Page_____ Of____

PROGRAM_____

PROGRAMMER_____

DATE_____

DOUBLE - PRECISION ADD ROUTINE

ADDR	INSTRUCTIONS B1	B2	B3	LABEL	MNEMONIC	OPERAND	COMMENTS
200	D8				CLD		CLEAR DECIMAL MODE FLAG (MODE = 0)
201	18				CLC		CLEAR CARRY FLAG (CARRY=0)
202	AD	02	03		LDA	L1	LOAD LOW ORDER BYTE, FIRST NUMBER
205	6D	04	03		ADC	L2	ADD WITH CARRY, LOW ORDER BYTE, SECOND NUMBER
208	8D	06	03		STA	L3	STORE LOW ORDER BYTE, RESULT
20B	AD	01	03		LDA	H1	LOAD HIGH ORDER BYTE, FIRST NUMBER
20E	6D	03	03		ADC	H2	ADD WITH CARRY, HIGH ORDER BYTE, SECOND NUMBER
211	8D	05	03		STA	H3	STORE HIGH ORDER BYTE, RESULT
214	4C	00	80		JMP	START	BRANCH TO MONITOR
301				H1	=	$301	HIGH ORDER BYTE OF FIRST NUMBER
302				L1	=	$302	LOW ORDER BYTE OF FIRST NUMBER
303				H2	=	$303	HIGH ORDER BYTE OF SECOND NUMBER
304				L2	=	$304	LOW ORDER BYTE OF SECOND NUMBER
305				H3	=	$305	HIGH ORDER BYTE OF RESULT
306				L3	=	$306	LOW ORDER BYTE OF RESULT
8000				START	=	$8000	MONITOR

The program is now entered. Examine each location to make sure that all values are correct. Then store the addend values in locations 0301-0304 as shown below. We'll use the numbers that were used in the example in Section 6.2.1, 0180 (Hex) and 0080 (Hex).

YOU KEY IN	DISPLAY SHOWS	EXPLANATION
(MEM) 301 (CR)	0301.**.	
01	0302.**.	Enter high order byte, first addend
80	0303.**.	Enter low order byte, first addend
00	0304.**.	Enter high order byte, second addend
80	0305.**.	Enter low order byte, second addend
(CR)	305.**..	

To execute the program, enter the command shown below.

YOU KEY IN	DISPLAY SHOWS	EXPLANATION
(GO) 200 (CR)	g 200.	Execute program starting at location 0200.

Now use MEM to examine locations 0305 and 0306. Verify that they are high and low order bytes of the result, 02 and 00. If you find other data at these locations, you will be pleased to know that the next section of this chapter tells you how to debug the program.

6.2.4 Debugging Methods

The first step in debugging is to make sure that the program and data have been entered correctly. Use the MEM command to examine the program starting address, and use the right-pointing arrow key to advance one location at a time and verify that the contents of each are correct. If you have a terminal, you can generate a listing by entering an SP command without turning on the tape punch or by using the VER command. Also examine the locations that contain the initial data.

If the program and data are correct, but the program still does not execute properly, you may want to use the SYM-1 DEBUG function. If DEBUG is ON when the execute (GO) command is entered, the program will execute the first instruction, then return control to the monitor. The address on the display will be the address of the first byte of the next instruction. If you again press GO (CR) to execute (do not specify an address this time), the computer will execute the next instruction, then halt as before. The program may be executed one step at a time in this manner.

By entering a non-zero Trace Velocity (at location A656), execution will automatically resume after a pause during which the Accumulator is displayed. Depress any key to halt automatic resumption.

After certain instructions, you will want to examine the contents of memory locations or registers. Use the MEM or REG commands, then resume operation by entering another GO command.

To examine the Carry Flag after the low order addition, for example, use the REG command as shown below.

YOU KEY IN	DISPLAY SHOWS	EXPLANATION
(ON)	unimportant	Turn DEBUG function ON
(GO) 200 (CR)	0201.2 .	Execute D8 instruction
(GO) (CR)	0202.2 .	Execute 18 instruction
(GO) (CR)	0205.2 .	Execute AD instruction
(GO) (CR)	0208.2 .	Execute 6D instruction, low order add with carry
(REG) (CR)	P 0208.	Program Counter
	r1 Fd.	Stack pointer
	r2 63.	Status register
(CR)	2 63.	End register examination
(GO) (CR)	020B.2 .	Execute 8D instruction
	.	
	.	

The Carry Flag is the lowest (rightmost) bit of the Status Register. To determine whether the flag was set, convert the Hex digits 63 to binary. The result of this conversion is 0110 0011, and since the low bit is "1", this confirms that the sum of the two low order bytes was greater than 255 (FF Hex).

You may turn the DEBUG switch OFF after any instruction. When you next press GO, the program will finish executing.

Since reading from or writing to any I/O port is the same as reading from or writing to a memory location, the DEBUG feature may also be used to debug I/O operations. When the port address is examined with a MEM command, the two Hex digits that represent data indicate the status of each line of the port. For example, if the value C2 is displayed, pin status is as follows:

PIN	7	6	5	4	3	2	1	0
STATUS	1	1	0	0	0	0	1	0

0 = Low
1 = High

For more advanced debugging techniques, including how to write and use your own trace routines, see Sections 9.5 and 9.6.

You now know how to code, enter, and debug programs on the SYM-1. Let's go look at two more examples that illustrate useful programming concepts.

6.3 CONDITIONAL TESTING

Most useful computer programs don't go in straight lines -- they don't simply execute a series of instructions in consecutive memory locations. They do perform different operations for different data by testing data words and jumping to different locations depending on the results of the test. Typical tests answer the following kinds of questions:

1. Is a selected bit of a specified data word a 1 or a 0?
2. Is a specified data word set to a selected ASCII character or numeric value?

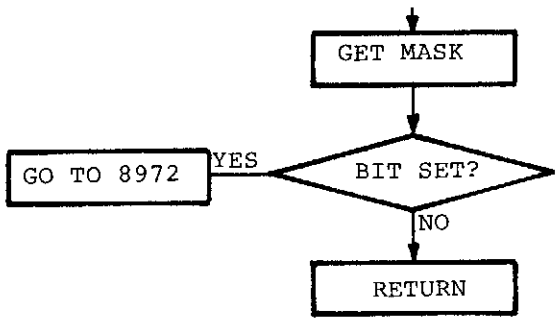
The sample program discussed below will answer question "1". It can be patched easily to answer question "2". You can use the principles you learn in the first two examples to make many more complicated tests.

Bit Testing

This sample program looks at the word in Hex location 31 and tests bit 3. If bit 3 is set to one, it jumps to location 8972; if bit 3 is zero, it returns to the executive. Location 8972 is a monitor subroutine that makes the SYM-1 go "beep".

The only problem involved is in isolating bit 3. The simplest way is to use a mask -- a word in memory with bit 3 set and none other. If we logically AND the mask with the sample word, the resultant value will be zero if bit 3 was zero and non-zero otherwise. The BIT test performs the AND and tests the value without altering the state of the accumulator.

Here is the flow chart. The code is in Figure 6-5. The mask (08 Hex) is in location 30, the test value in location 31.



Hint

If you wish to test bit 0 or bit 7 of a byte, you need not use a mask. Simply use a shift operation to place the selected bit in the CARRY status bit and use a BCC or BCS to test CARRY. This saves one or more program locations. Note that it alters the accumulator - you may have to shift it back for later processing.

Character, Value, or Magnitude Testing

To test whether a byte is exactly equal to an ASCII character or a value, use the Compare command or first set a mask location exactly equal to the character or value. Then use the EOR command to find the exclusive OR of the two values and test the result for zero. It will be zero if and only if the values were identical. Note that this destroys the test value -- keep another copy of it if you must use it again.

To test whether a byte is greater, equal to, or less than a given value, use the Compare command or set a mask to the test values and subtract it from the test value. The test value will be destroyed. Test the result to see whether it is positive, negative, or zero (this takes two sequential tests) and skip accordingly. Try writing a program that makes a series of magnitude tests to determine whether a given byte is an ASCII control character (0-1F Hex), punctuation mark, number, or letter. The values of the ASCII character set are listed on the summary instruction card.

SYNERTEK SYSTEMS CORPORATION
2589 Scott Boulevard
Santa Clara, California 95050
(408) 247-8940

PROGRAM _____

PROGRAMMER _____

DATE _____

BIT 3 TEST ROUTINE

ADDR	INSTRUCTIONS B1	B2	B3	LABEL	MNEMONIC	OPERAND	COMMENTS
30	08			MASK			BIT 3 MASK
31	XX			TEST			(USE DIFFERENT VALUES)
200	A5	30			LDA	MASK	GET MASK
202	24	31			BIT	TEST	COMPARE AND SET ZERO BIT
204	FØ	03			BEQ	HOME	NO BIT, RETURN
206	4C	72	89		JMP	BEEP	BEEP ANNUNCIATOR
209	6Ø			HOME	RTS		RETURN

6.4 MULTIPLICATION

The sample program described here multiplies two one-byte unsigned integers and stores the results in two bytes. Note that in any base of two or more, the product of two numbers may be as long as the sum of the lengths of the numbers. In decimal, 99 X 99= 9801; in Hex FF X FF= FE01.

Since many programs will involve multiplication, it is not good practice to write a multiplication routine every time the need comes up. The sample is set up as a subroutine to allow it to be used by many programs. Serious programmers will usually wind up with libraries of subroutines specialized for their applications.

How to Multiply

Multiplication is normally introduced to students as a form of sequential addition. Humans can in fact multiply 22 (decimal) by 13 by performing an addition:

22 + 22 + 22 . . . . 22 = 286

This technique is of course foolish -- it involves a lot of work and a high probability of error. It would be easy to write a program that would multiply this way (try it) but it would be a terrible waste of time.

How then to multiply? We could use a table. Humans use memorized tables that work up to about 10 X 10:

7 X 8 = 56

Humans cannot, however, remember well enough to know that:

22 X 13 = 286

Computers, of course, can "remember" an arbitrarily large table. But the table for the problem at hand would have FFFF entries, which is far too many for practicality.

Humans solve the problem by breaking the multiplication down into smaller steps. We multiply one factor, one digit at a time, by each digit of the other factor in turn. Then we shift some of the partial products to the left and add:

```
    22
X   13
   ---
    66
   22
  ----
   286
```

We would multiply the binary equivalents of the numbers the same way:

```
   10110
    1101
  -------
   10110
      0
  10110
 10110
---------
100011110
```

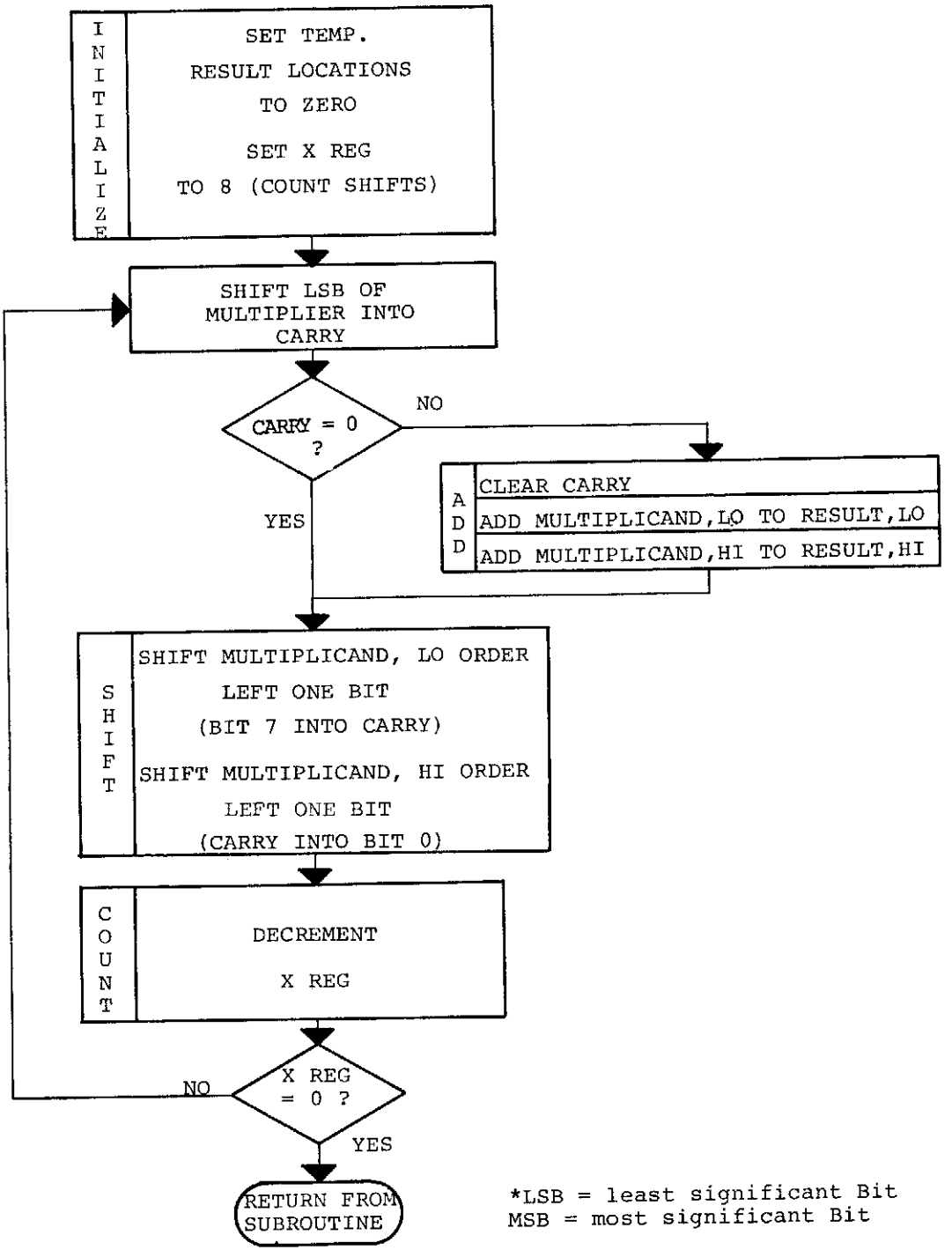


Figure 6-6. GENERAL MULTIPLICATION FLOWCHART

A little figuring will verify that the result is correct. Note that the "tables" for multiplying binary numbers by a single digit are very simple -- a number times one is itself; a number times zero is zero. We can multiply, then, by using a series of additions and shifts, as shown in the flow chart below. The first factor is eight bits long; the second is extended to two bytes (the high-order byte is zero), and the result goes into two bytes set initially to zero. The flowchart in Figure 6-6 is general and not suitable for direct coding.

This procedure could be coded quite easily. Each bit test on the first factor could be made with a different mask as shown in the previous example. Note, however, that the same basic set of instructions is repeated eight times, wasting memory space. A more efficient routine would loop over the same code eight times.

The more efficient routine could also use eight masks, but there's a simpler way. Simply shift the factor to the left once per addition. The bit to be tested will wind up in the CARRY indicator, and we can simply test that. Figure 6-7 is a more formal flowchart of the multiply routine as it is coded that it includes the coding details. The coding chart is shown in Figure 6-8.

Testing

The listing below shows one way to key in the program. The code occupies the RAM space from 200 to 222 Hex. The factor come from locations 21 and 22; the product goes to locations 23 and 24.

Note that the original factors are destroyed by the routine. If it is necessary to preserve them for other subroutines, simply copy them into unused memory locations and perform the multiplication on the copies.

Division

Try to write a parallel routine for performing integer division that divides a two-byte quotient and a two-byte remainder. You may wish to test the remainder and, if its MSB is one, round the result by incrementing the quotient.

Arithmetic

The examples given so far show some basic integer arithmetic techniques. They may be expanded easily for double-precision operation. (Multiply two bytes by two bytes for a four-bit product. Use double-precision addition and fifteen shifts instead of seven.)

MULTIPLIER = P

MULTIPLICAND = Q

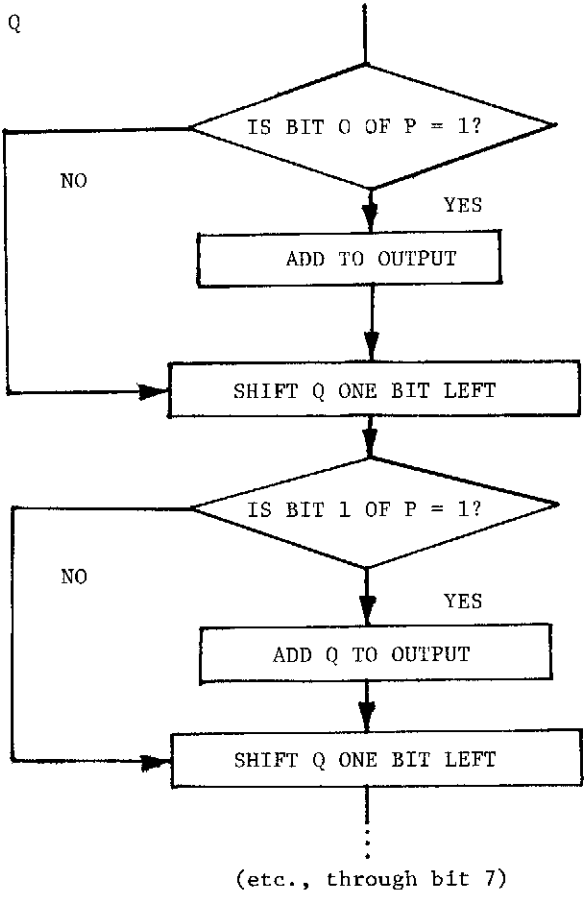


Figure 6-7. DETAILED MULTIPLICATION FLOWCHART

SYNERTEK SYSTEMS CORPORATION

PROGRAM _____

PROGRAMMER _____

DATE _____

SINGLE - PRECISION MULTIPLY ROUTINE

ADDR	INSTRUCTIONS B1	B2	B3	LABEL	MNEMONIC	OPERAND	COMMENTS
200	A9	00		MULT1	LDA	#0	ZERO ACCUMULATOR
202	85	20			STA	INHI (20)	SET TEMPORARY STORAGE LOCATION (20) TO ZERO
204	85	23			STA	OUTLO (23)	" LOW BYTE RESULT " (23) " "
206	85	24			STA	OUTHI (24)	" HIGH " " " (24) " "
208	A2	08			LDX	#8	SET X TO 8 TO COUNT SHIFTS
20A	46	22		MORE	LSR	IN2 (22)	SHIFT FACTOR RIGHT
20C	90	0D			BCC	ZERBIT (1D)	IF CARRY = 0 SKIP ADDITION, GO TO ZERBIT
20E	18				CLC		CLEAR CARRY
20F	A5	23			LDA	OUTLO	GET LOW BYTE ASSEMBLED SO FAR
211	65	21			ADC	IN1	ADD CURRENT TERM
213	85	23			STA	OUTLO	SAVE UPDATED LOW BYTE
215	A5	24			LDA	OUTHI	GET HIGH BYTE ASSEMBLED SO FAR
217	65	20			ADC	TEMPHI	ADD CURRENT TERM
219	85	24			STA	OUTHI	SAVE UPDATED HI BYTE
21B	06	21		ZERBIT	ASL	IN1	SHIFT LEFT FOR NEXT ADDITION
21D	26	20			ROL	INHI	SHIFT HIGH BYTE LEFT (ENTER CARRY)
21F	CA				DEX		DECREMENT INDEX REGISTER (COUNT ADDS)
220	D0	E8			BNE	MORE	IF X > 0, GO BACK AND DO NEXT ADD
222	60				RTS		DONE; GO BACK TO CALLING ROUTINE

Figure 6-8. SINGLE-PRECISION MULTIPLY ROUTINE

CHAPTER 7

OSCILLOSCOPE OUTPUT FEATURE

7.1 INTRODUCTION

Your SYM-1 module is hardware-equipped to allow you to use an ordinary oscilloscope as a display device. In this section, we will describe the hardware and connections between the system and the oscilloscope and also provide a listing of a software driver for this output. This listing is just one way of handling the oscilloscope output; you may wish to modify it or develop your own.

7.2 OPERATION OF OSCILLOSCOPE OUTPUT

The circuitry shown in the detail on the schematic (Figure 4-9) enables the SYM to output alphanumeric characters to an oscilloscope. The circuitry is adapted from a published schematic and was included on the SYM to help relieve the bottleneck found on most single-board computers, i.e., the 7 segment displays. Many things can be done with the scope-out circuit, like displaying alphanumeric characters, bar graphs, and game displays. The alphanumeric output is usually organized as 16 or 32 characters, each character being a 5-by-7 dot matrix. The characters could be English, Greek or Cuneiform, or could even be stick-men, cars, dog houses, or laser guns.

The "video" signal from the collector of Q10, is 3V peak-to-peak with a cycle time of about 50 ms (using the suggested software driver included in section 7.3). The sync pulse which begins the line should synchronize all triggered sweep scopes and most recurrent sweep scopes. In the driver which follows, sync could be brought out on a separate pin by replacing the code from SYNC to CHAR with a routine that would output a pulse on PB4 or some other output line.

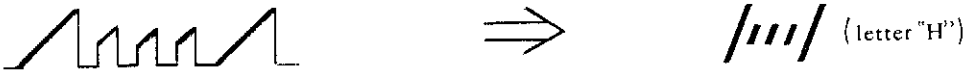
7.2.1 Connection Procedures

Connect the oscilloscope vertical input to pin R on connector AA ("scope out") and connect scope ground to pin 1 of connector AA (SYM ground). Start the software and adjust the scope for the stable 32-character display. If the sync pulse was output on PB4, connect the scope's trigger to pin 4 of connector AA.

7.2.2 Circuit Operation

The operation of the circuit is simple. Basically, the circuit is a sawtooth waveform generator whose output is sometimes the sawtooth and sometimes ground. The sawtooth is generated by the current source, Q9-Q17-R42-R43, charging C9. When C9 gets up to about 3V the discharge path, Q19-Q18-R41-R44, shorts it back to ground due to a pulse sent out by CA-2. The sawtooth waveform is shown below and forms the columns of the display.

By pulling the sawtooth to ground with Q10 any columns or portions thereof can be "removed" from the display. The result of this can be seen below:



The sawtooth is pulled to ground by bringing CB-2 high.

Because Q10 in the "ON" state will cause loading of C9 (thru R45) and C9 will charge a little more slowly, the time for a "dark" column should be slightly longer than for a "light" column.

If more than 8 vertical dots are desired, the charging rate of C9 must be slowed by lowering the charging current. R42 controls the charging current and can be increased up to about 10K before the loading effects of R45 get completely out of hand.

7.3 USING OUR SOFTWARE

The program listing in Table 7-1 is one way of handling oscilloscope output. After entering the program and character table and attaching an oscilloscope to the scope output, enter the following commands:

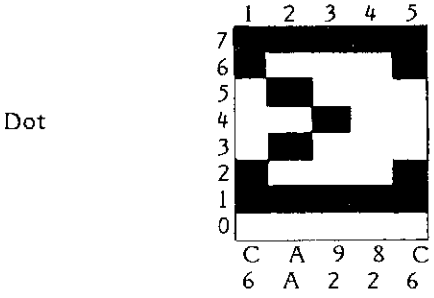
	Comments
.SD 500, A670**(CR)**	Change SCANVEC. (DISPLAY GOES BLANK)
.SD 58C, A664**(CR)**	Change OUTVEC.
.SD 560, A661**(CR)**	Change INVEC.

Now enter any stream of characters from the HKB to fill SCPBUF.

Put the scope input on AC couple and the trigger on DC couple. Adjust the time base, attenuation, and trigger until the display becomes readable. If your screen is very small, you may wish to change the number of characters per line by adjusting the value at location $0506.

Example: Creating translation table for scope driver.

Character: $\sum$ (Greek capital letter sigma)



Dot

Hex Equivalent

Each byte corresponds to a single column, with each bit corresponding to a single dot.

sigma = $C6, $AA, $92, $82, $C6

Bit Ø is always Ø to raise the character off of the Ground line.

Table 7-1. OSCILLOSCOPE OUTPUT DRIVER SOFTWARE LISTING

```
LINE # LOC      CODE        LINE

0002  0000                  # SCOPE LINE DRIVER 05/01/78
0003  0000                  # USES CHARACTER SET IN TABLE SYMBLS
0004  0000                  # 5 BYTES PER CHAR
0005  0000                  # ENTRY 'LINE' IS ANALOGOUS TO 'SCAND'
0006  0000                  # BELOW ROUTINES HKEY AND HDOUT INTERFACE TO HEX KB
0007  0000                  # CHAR SET PROVIDED IS FOR HEX KB
0008  0000                  # AND RELATED TO ASCII TABLE IN MONITOR ROM
0009  0000                  # THIS DRIVER CAN ACCESS A MAX OF 51 CHARS
0010  0000                  TXTSHV =$8A06
0011  0000                  SCNVEC =$A66F
0012  0000                  GETKEY =$88AF
0013  0000                  KEYQ   =$8923
0014  0000                  SAVER  =$8188
0015  0000                  BEEPP3 =$8975
0016  0000                  ASCIM1 =$88EE
0017  0000                  RESALL =$81C4
0018  0000                  PCR3   =$AC0C          #CA2,CB2 = SCOPE
0019  0000                  TXTCTR =$03FE
0020  0000                  COLCTR =$03FF
0021  0000                  TEXT   =$A600          #SCOPE BUFFER IN SYS RAM
0022  0000                  SYMBLS =$0400          #CHARACTER TABLE
0023  0000                         *=$500
0024  0500  A9 EE           LINE   LDA #$EE        #DISCHARGE CAP
0025  0502  8D 0C AC               STA PCR3
0026  0505  A9 21                  LDA #32+1       ## CHARS PER LINE
0027  0507  8D FE 03               STA TXTCTR
0028  050A  A9 CC           SYNC   LDA #$CC        #CHARGE CAP FOR SYNC
0029  050C  8D 0C AC               STA PCR3
0030  050F  A2 EA           LDLY   LDX #$EA        #LONG DELAY !
0031  0511  CA                     DEX
0032  0512  D0 FC                  BNE LDLY+1
0033  0514  CE FE 03         CHAR   DEC TXTCTR      #LOOP HERE FOR CHAR
0034  0517  AE FE 03               LDX TXTCTR
0035  051A  D0 03                  BNE POIMFG
0036  051C  4C 23 89         EXIT   JMP KEYQ        #SCAN KB AND RETURN
0037  051F                  #
0038  051F  BD FF A5         POIMFG LDA TEXT-1,X    #POINTER MANUFACTURER
0039  0522  0A                     ASL A            #PTR X 4 + PTR
0040  0523  0A                     ASL A
0041  0524  18                     CLC
0042  0525  7D FF A5               ADC TEXT-1,X     #MULT'PY BY 5
0043  0528  AA                     TAX
0044  0529  A9 06                  LDA #6
0045  052B  8D FF 03               STA COLCTR
0046  052E  A9 EE           COLUMN LDA #$EE        #LOOP HERE FOR COL'S
0047  0530  8D 0C AC               STA PCR3         #DISCHARGE CAP
0048  0533  CE FF 03               DEC COLCTR
0049  0536  30 DC                  BMI CHAR         #BRANCH IF DONE W/6 COL'S
0050  0538  D0 02                  BNE COLUP
0051  053A  A2 00                  LDX #0           #INTER CHAR SPACE
0052  053C  A9 EC           COLUP  LDA #$EC        #START RAMP UP ...
0053  053E  8D 0C AC               STA PCR3         #... BUT HOLD DOT DOWN
0054  0541  E8                     INX              #NEXT COL
0055  0542  8A                     TXA              #SAVE X
0056  0543  48                     PHA
```

Table 7-1. OSCILLOSCOPE OUTPUT DRIVER SOFTWARE LISTING (Continued)

```
LINE #  LOC   CODE        LINE
0057  0544  BD FF 03          LDA SYMBLS-1,X  ;GET COL
0058  0547  A0 08             LDY #8          ;COUNT DOTS
0059  0549  88          DOT   DEY
0060  054A  30 0F             BMI CLEAN
0061  054C  4A                LSR A           ;NEXT DOT IN CARRY
0062  054D  B0 04             BCS LIGHT       ;C SET = LIGHT, C CLEAR = DARK
0063  054F  A2 EC       DARK  LDX #$EC        ;PULL OUTPUT LOW
0064  0551  D0 02             BNE *+4
0065  0553  A2 CC       LIGHT LDX #$CC        ;OUTPUT FOLLWS RAMP UP
0066  0555  8E 0C AC           STX PCR3
0067  0558  4C 49 05           JMP DOT
0068  055B  68          CLEAN PLA             ;RESTORE X
0069  055C  AA                TAX
0070  055D  4C 2E 05           JMP COLUMN
0071  0560                ;
0072  0560                ;
0073  0560  20 AF 88      HKEY  JSR GETKEY      ;GET KEY + ECHO TO SCOPE
0074  0563  20 88 81      SCPDSP JSR SAVER      ;FILL SCPBUF FROM ASCII IN A
0075  0566  29 7F               AND #$7F
0076  0568  C9 07               CMP #$07        ;BELL?
0077  056A  D0 03               BNE NBELL
0078  056C  4C 75 89            JMP BEEPP3
0079  056F                ; SEARCH ASCII TABLE IN MONITOR ROM
0080  056F  A2 36       NBELL LDX #$36
0081  0571  DD EE 8B     OUD2  CMP ASCIM1,X
0082  0574  F0 06               BEQ GOTX
0083  0576  CA                  DEX
0084  0577  D0 F8               BNE OUD2
0085  0579  4C C4 81            JMP RESALL      ;NOT IN TABLE
0086  057C  CA            GOTX  DEX
0087  057D  8A                  TXA
0088  057E  C9 0B               CMP #$0B        ;TABLE NOT CONTINUOUS
0089  0580  90 03               BCC GOOD
0090  0582  38                  SEC
0091  0583  E9 05               SBC #5          ;ADJUST DISCONTINUITY
0092  0585  CA            GOOD  DEX
0093  0586  20 06 8A           JSR TXTSHV      ;SHOVE SCPBUF DOWN
0094  0589  4C C4 81            JMP RESALL
0095  058C  20 63 05      HDOUT JSR SCPDSP      ;CHAR TO SCPBUF AND SINGLE SCAN
0096  058F  4C 6F A6            JMP SCNVEC
0097  0592                      .END
```

Table 7-1. OSCILLOSCOPE OUTPUT DRIVER SOFTWARE LISTING (Continued)

```
;
; 8X5 MATRIX CHAR SET FOR SCOPE LINE DRIVER
; CONTAINS ALL HEX KB CHARS
; FIRST BYTE OF TABLE MUST BE 00
; EACH CHAR : FIRST BYTE = LEFTMOST COLUMN,
;             MSB = TOP DOT, LSB = 0, BIT 1 = BOTTOM DOT
;
*=$400 ;PAGE 4 ALLOCATED TO CHARACTER SET
.BYT $00,$7C,$92,$A2,$7C ;ZERO
.BYT $00,$42,$FE,$02,$00 ;ONE
.BYT $4E,$92,$92,$92,$62 ;TWO
.BYT $44,$82,$92,$92,$6C ;THREE
.BYT $18,$28,$48,$FE,$08 ;FOUR
.BYT $E4,$A2,$A2,$A2,$9C ;FIVE
.BYT $3C,$52,$92,$92,$0C ;SIX
.BYT $86,$88,$90,$A0,$C0 ;SEVEN
.BYT $6C,$92,$92,$92,$6C ;EIGHT
.BYT $60,$92,$92,$94,$78 ;NINE
.BYT $3E,$50,$90,$50,$3E ;A
.BYT $00,$1E,$86,$4A,$32 ;C/R
.BYT $10,$10,$10,$10,$10 ;DASH
.BYT $82,$44,$28,$10,$00 ;RIGHT ARROW
.BYT $FE,$FE,$FE,$FE,$FE ;SH
.BYT $7C,$82,$82,$8A,$4E ;G
.BYT $FE,$90,$98,$94,$62 ;R
.BYT $FE,$40,$30,$40,$FE ;M
.BYT $FE,$02,$02,$02,$02 ;L2
.BYT $44,$A2,$92,$8A,$44 ;S2
.BYT $80,$80,$80,$80,$80 ;U0
.BYT $02,$02,$02,$02,$02 ;U1
.BYT $82,$82,$82,$82,$82 ;U2
.BYT $FE,$00,$00,$00,$00 ;U3
.BYT $FE,$00,$00,$00,$FE ;U4
.BYT $1E,$12,$12,$12,$1E ;U5
.BYT $F0,$90,$90,$90,$F0 ;U6
.BYT $80,$80,$80,$80,$F0 ;U7
.BYT $04,$02,$02,$02,$FC ;J
.BYT $E0,$18,$06,$18,$E0 ;V
.BYT $FF,$FF,$FF,$FF,$FF ;ASCII
.BYT $FE,$92,$92,$92,$6C ;B
.BYT $7C,$82,$82,$82,$44 ;C
.BYT $FE,$82,$82,$82,$7C ;D
.BYT $FE,$92,$92,$82,$82 ;E
.BYT $FE,$90,$90,$80,$80 ;F
.BYT $44,$A2,$92,$8A,$44 ;SD
.BYT $10,$10,$7C,$10,$10 ;+
.BYT $00,$10,$28,$44,$82 ;<
.BYT $00,$00,$00,$00,$00 ;SH
.BYT $FE,$02,$02,$02,$02 ;LP
.BYT $44,$A2,$92,$8A,$44 ;SP
.BYT $FE,$04,$08,$04,$FE ;W
.BYT $FE,$02,$02,$02,$02 ;L1
.BYT $44,$A2,$92,$8A,$44 ;S1
.BYT $00,$06,$06,$00,$00 ;DECIMAL
.BYT $00,$00,$00,$00,$00 ;BLANK
.BYT $40,$80,$8A,$90,$60 ;QUESTION
.BYT $FE,$90,$90,$90,$60 ;P
.END
```

CHAPTER 8

SYSTEM EXPANSION

This chapter discusses the means by which you can expand your SYM-1 microcomputer system by adding memory and peripheral devices to its basic configuration. By now, you realize that data access, whether from RAM, PROM or ROM is a function of addressing interface devices (i.e., 6522's and 6532). Hardware has been built into your SYM-1 module to allow large-scale expansion of the system. A thorough understanding of the SYM-1 System Memory Map (Figure 4-10) will aid considerably in understanding how to expand your system.

8.1 MEMORY EXPANSION

Your SYM-1 module comes equipped with 1K of on-board RAM. It also contains all address decoding logic required to support an additional 3K on-board with no changes by you. In other words, to add 3K of on-board RAM, all you need to do is purchase additional SY2114 devices and plug them into the sockets provided on your board. Your PC board is marked for easy identification of 1K memory blocks. R0 equals the lower 1K block and R3 equals the upper 1K block. LO means low order data lines (D0-D3) and HI means high order data lines (D4-D7).

You will recall that the lowest 8K memory locations are defined by an address decoder included on your SYM-1 module (a 74LS138). The eight outputs of this decoder ($\overline{00}$-$\overline{1C}$) each define a 1K block of addresses in the lowest 8K of the Memory Map. Four of the outputs ($\overline{00}$, $\overline{04}$, $\overline{08}$, $\overline{0C}$) are used to select the on-board static RAM. The remaining four outputs ($\overline{10}$, $\overline{14}$, $\overline{18}$, $\overline{1C}$) are used to interface to the Application Connector (Connector "A"), where you can use them to add another 4K of off-board memory. Again, no external decoding logic is required. By this simple means, you can convert your SYM-1 module into an 8K device quickly. Figure 8-1 shows you how to interface these decode lines at the connector for your SYM-1 system.

To go beyond this 8K size, conceivably up to the maximum 65K addressability limit of the SYM CPU, you could build or buy an additional memory board with on-board decoding logic. In this case, you will use the Expansion Connector (Connector "E") in a manner shown schematically in Figure 8-2. Note that the three high-order address bits (AB13-AB15) not used in the earlier expansion are brought to this connector as shown. These are then used with a decoder to create outputs $\overline{M0}$ through $\overline{M7}$, which in turn are used to select and de-select additional decoders (line receivers). You need add only as many decoders (one for each 8K block of memory) as you need for the expansion you require.

Incidentally, the line receivers shown in Figure 8-2 are provided for electrical reasons. There are loading limitations on the address bus lines of the 6502 CPU, which require the insertion of these receivers. (For your information, each 6502 address line is capable of driving one standard TTL load and 130pf of capacity.)

You should make a careful study of the loading limitations of the required SYM-1 lines before deciding on memory expansion size and devices. It is likely you will want to use additional buffer circuits to attain "cleaner" operation of your expanded memory in conjunction with your SYM-1 system.

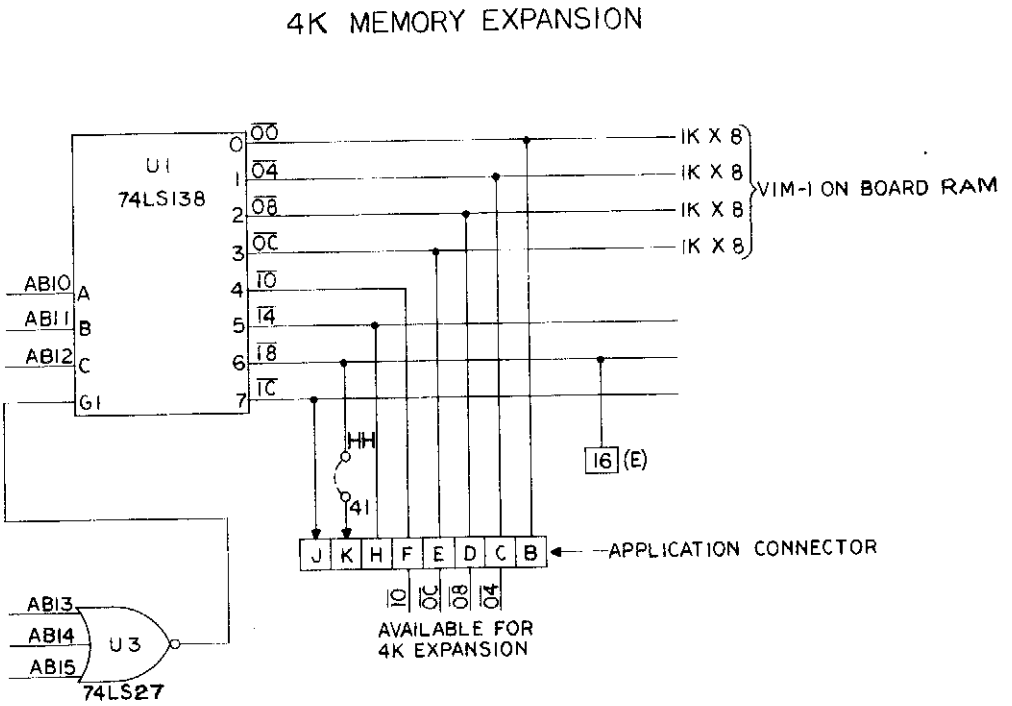


Figure 8-1. 4K MEMORY EXPANSION

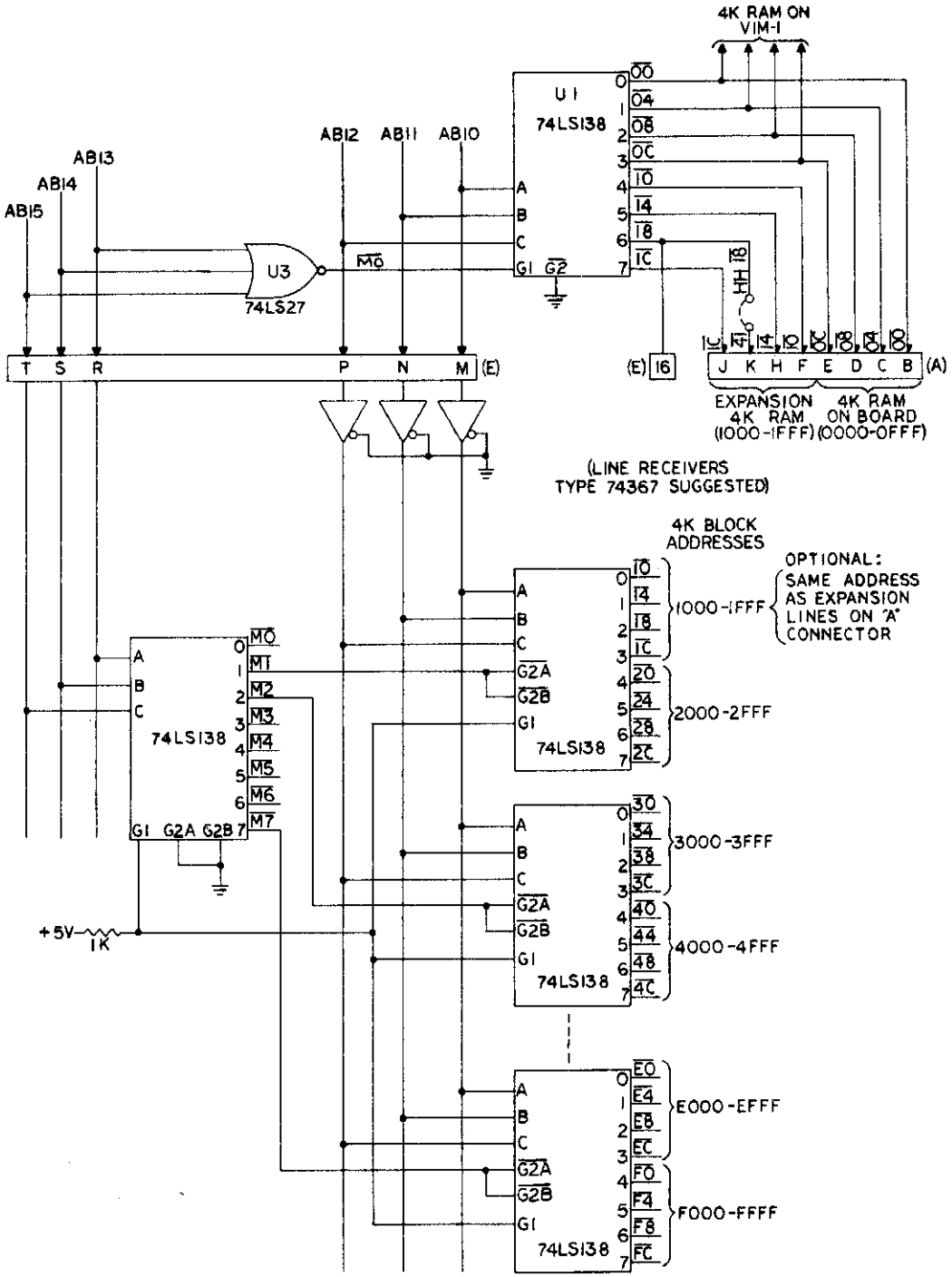


Figure 8-2. MEMORY - I/O EXPANSION TO 65K

8.2 PERIPHERAL EXPANSION

As you already know, the SYM-1 microcomputer system includes 51 I/O lines. This means, theoretically, that you could drive as many as 51 peripheral lines (plus 4 control lines) with your SYM-1.

Using either Application Connector ("A" or "AA"), you can add most commercially available printers or other devices requiring parallel interfaces, although you will have to create your own software driver for the printer. Since the provision of that driver is, to some extent, dependent upon the printer you purchase, we do not attempt to discuss the implementation of the software in this manual.

You can expand your SYM-1 system's peripheral I/O capability easily and quickly merely by installing an additional SY6522 in the socket provided for that device. This will give you 16 additional on-board data lines with no requirement for additional work (beyond the software driver) on your part. To go beyond that level, you must use the Expansion Port (Connector "E") described earlier.

Again, we emphasize that the proper understanding and use of the Memory Map in Figure 4-10 will allow you to use your imagination in expanding the I/O capability of your SYM-1 system. Its flexibility is extremely broad and the fact that all I/O and memory are handled as an addressing function allows you expandability to the full capability of the 6502 CPU itself.

CHAPTER 9

ADVANCED MONITOR AND PROGRAMMING TECHNIQUES

This chapter contains information which you will find useful as you explore the more sophisticated capabilities of your versatile SYM-1 microcomputer system. As we have pointed out many times, the SYM-1 is the most flexible and expandable monitor of its kind. The SUPERMON monitor uses transfer vectors and other techniques to allow you to modify its operation, and these are provided in detail in this chapter. In addition, the extended use of debug and trace facilities, which are invaluable tools as your programming skill advances, are explained. The use of the Hex keyboard provided on your SYM-1 for configurations using a printer (or other serial device) without a keyboard is also described. And last, an example and discussion of extending SUPERMON's command repertoire.

9.1 MONITOR FLOW

SUPERMON is the 4K byte monitor program supplied with your SYM-1. It resides in locations 8000-8FFF on a single ROM chip. It shares the stack with user programs and uses locations 00F8-00FF in Page Zero. In addition, it uses locations A600-A67F (RAM on the 6532), which are referred to as 'System RAM'. Since these locations are dedicated to monitor functions SUPERMON write protects them before transferring control to user programs.

The flowcharts in Figures 9-1 through 9-5 will demonstrate the major structure of SUPERMON. You will notice that GETCOM (and its entry, PARM), DISPAT, and ERMSG are subroutines, and therefore available for your programs' use. Note that a JSR to ACCESS to remove write protection from System RAM is necessary before using most monitor routines. Also, notice that the unrecognized command flow (error) is vectored. Thus, you can extend the monitor with your own software.

9.2 MONITOR CALLS

SUPERMON contains many subroutines and entry points which you will want to use in order to save memory and code and avoid duplication of effort. Table 9-1 is a summary of calls and their addresses.

The three calls which you will most commonly use are:

```
JSR   ACCESS   (address 8B86) (must be called before using LED display)
JSR   INCHR    (address 8A1B)
JSR   OUTCHR   (address 8A47)
```

ACCESS is used to unwrite-protect system RAM. In performing the input/output, these routines save all registers and use INVEC and OUTVEC, so all you need be concerned with when using them are the ASCII characters passed as arguments in the accumulator.

9.3 MONITOR CALLS, ENTRIES AND TABLES

Table 9-1, which occupies the next several pages of this Chapter, provides you with a comprehensive list of important subroutine symbolic names, addresses, registers and functions of SUPERMON monitor calls, entry points and tables. With this data, you can more easily utilize SUPERMON to perform a wide variety of tasks. All (except those marked with an asterisk) are callable by JSR.

Table 9-1. MONITOR CALLS, ENTRIES AND TABLES

NAME	ADDRESS	REGISTERS ALTERED	FUNCTION (S)
*MONITR	8000		Cold entry to monitor. Stack, D flag initialized, System RAM unprotected.
*WARM	8003		Warm entry to monitor
USRENT	8035		User pseudo-interrupt entry - saves all registers when entered with JSR. Displays PC and code 3. Passes control to monitor.
SAVINT	8064	ALL	Saves registers when called after interrupt. Returns by RTS.
DBOFF	80D3	A,F	Simulates depressing debug off key.
DBON	80E4	A,F	Simulates depressing debug on key.
DBNEW	80F6	A,F	Release debug mode to key control.
GETCOM	80FF	A,F	Get command and 0-3 parameters. No error: A=0D (carriage return) Error: A contains erroneous entry.
DISPAT	814A	A,F	Dispatch to execute blocks. Dispatch to URCVEC if error. At return, if error: Carry set, A contains byte in error.
ERMSG	8171	F	If Carry set, print (CR)ER NN, where NN is contents of A.
SAVER	8188	None	Save all registers on stack. At return, stack looks like: F (See paragraph 9.9) A X Y
*RESXAF	81B8	restored	Jumped to after SAVER, restore registers from stack except A,F unchanged, perform RTS.
*RESXF	81BE	restored	Jumped to after SAVER, restore registers from stack except F unchanged, perform RTS.
*RESALL	81C4	restored	Jumped to after SAVER, restore all registers from stack, perform RTS.
INBYTE	81D9	A,F	Get 2 ASCII Hex digits from INCHR and pack to byte in A. If Carry set, V clear, first digit non-Hex. If Carry set, V set, second digit nonHex. N and Z reflect compare with carriage return if Carry set.

*Do not enter by JSR.

Table 9-1. MONITOR CALLS, ENTRIES AND TABLES (Continued)

NAME	ADDRESS	REGISTERS	FUNCTION (S)
PSHOVE	8208	X,F	Shove Parms down 16 bits; Move: P2 to P1 P3 to P2 zeros to P3
PARM	8220	A,F	Get 0 to 3 parameters. Return on (CR) or error. A contains last character entered. Flags reflect compare with (CR).
ASCN1B	8275	A,F	Convert ASCII character in A to 4 bits in LO nibble of A. Carry set if non-Hex.
OUTPC	82EE	A,X,F	Print user PC. At return, A=PCL, X=PCH.
OUTXAH	82F4	F	Print X,A (4 Hex digits)
OUTBYT	82FA	F	Print A (2 Hex digits)
NIBASC	8309	A,F	Convert LO nibble of A to ASCII Hex in A.
COMMA	833A	F	Print comma.
CRLF	834D	F	Print **(CR) (LF).**
DELAY	835A	F,X	Delay according to TV. (Relation is approximately logarithmic, base=2). Result of INSTAT returned in Carry.
INSTAT	8386	F	If key down, wait for release. Carry set if key down. (Vectored thru INSVEC)
GETKEY	88AF	A,F	Get key from Hex keyboard (more than one if SHIFT or ASCII key used) return with ASCII or HASH code in A. Scans display while waiting (vectored through SCNVEC).
HDOUT	8900	A,X,Y,F	ASCII character from A to Hex display, scan display once, return with Z=1 if key down.
SCAND	8906	A,X,Y,F	Scan the LED display once from the data in DISBUF. Return Z set if a key on hex keyboard is down.
KEYQ	8923	A,F	Determine if key down on Hex keyboard. If down, then Z=1.
KYSTAT	896A	A,F	Determine if key down. If down, then Carry set.
BEEP	8972	None	BEEP on-board beeper.
HKEY	89BE	A,F	Get key from Hex keyboard and echo in DISBUF. ASCII returned in A. Scans display while waiting (vectored thru SCNVEC)

*Do not enter by JSR

Table 9-1. MONITOR CALLS, ENTRIES AND TABLES (Continued)

NAME	ADDRESS	REGISTERS	FUNCTION (S)
OUTDSP	89C1	None	Convert ASCII in A to segment code, put in DISBUF.
TEXT	8A06	F	Shove scope buffer down, push A onto SCPBUF.
INCHR	8A1B	A,F	Get character (vectored thru INVEC). Drop parity, convert to upper case. If character CTL O (0F), toggle Bit 6 of TECHO and get another.
NBASOC	8A44	A,F	Convert low nibble of A to ASCII, output (vectored thru OUTVEC).
OUTCHR	8A47	None	Output ASCII from A (vectored thru OUTVEC). Output inhibited by Bit 6 of TECHO.
INTCHR	8A58	A,F	Get character from serial ports. Echo inhibited by Bit 7 of TECHO. Baud rate determined by SDBYT. Input, echo masked with TOUTFL.
TSTAT	8B3C	A,F	See if break key down on terminal. If down, then Carry set.
*RESET	8B4A	All	Initialize all registers, disable POR, stop tape, initialize system RAM to default values, determine input on keyboard or terminal, determine baud rate, cold monitor entry.
*NEWDEV	8B64		Determine baud rate, cold monitor entry.
ACCESS	8B86	None	Un-write protect System RAM.
NACCESS	8B9C	None	Write protect System RAM.
*TTY	8BA7	A,X,F	Set vectors, TOUTFL, and SDBYT for TTY.
*DFTBLK	8FA0	Table	Default block - entirely copied into System RAM (A620 - A67F) at reset.
*ASCII	8BEF	Table	Table of ASCII codes and HASH codes.
*SEGS	8C29	Table	Table of segment codes corresponding to ASCII codes (above).

*Do not enter by JSR

MAIN MONITOR FLOW

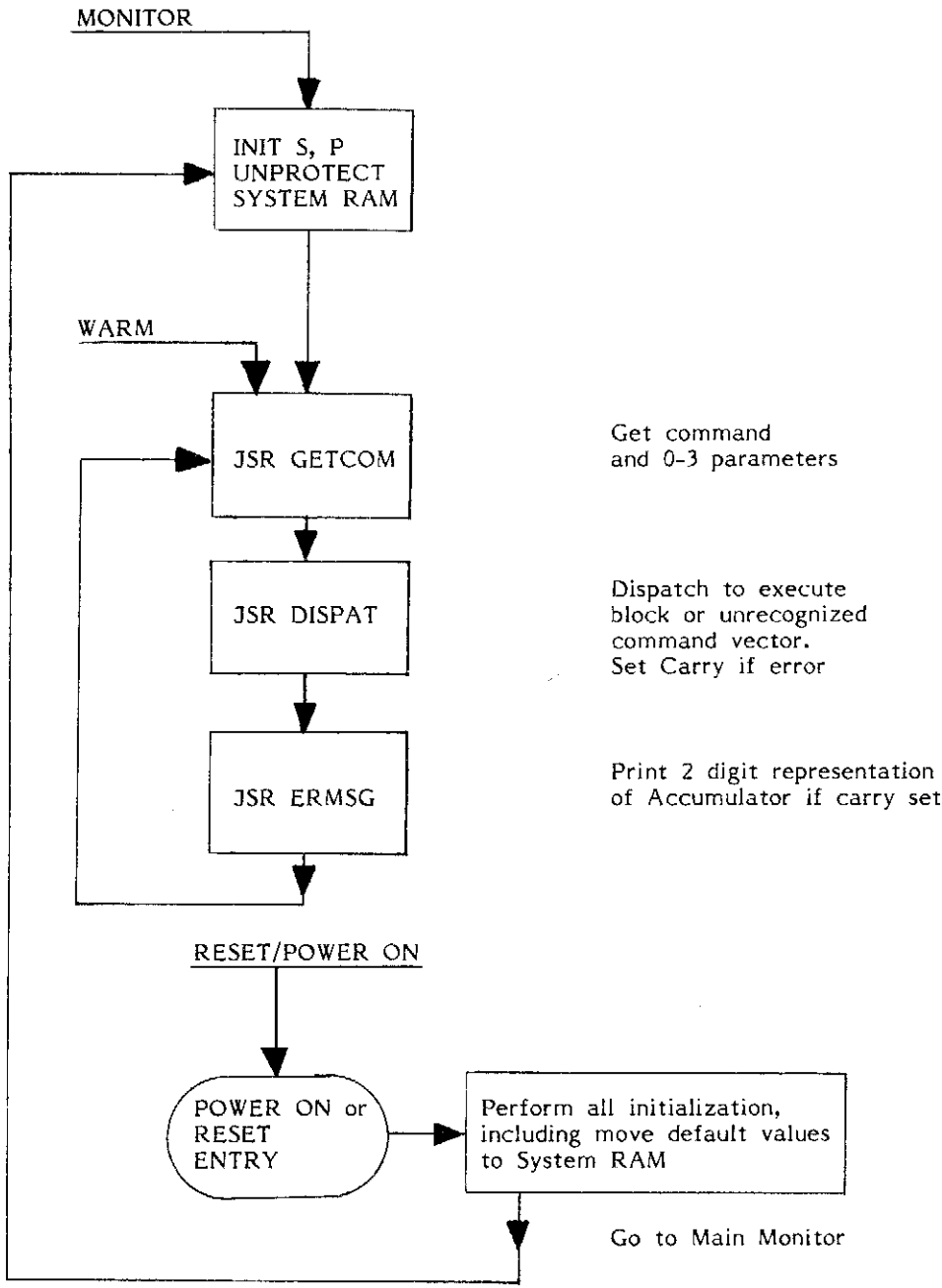


Figure 9-1. MAIN MONITOR FLOW

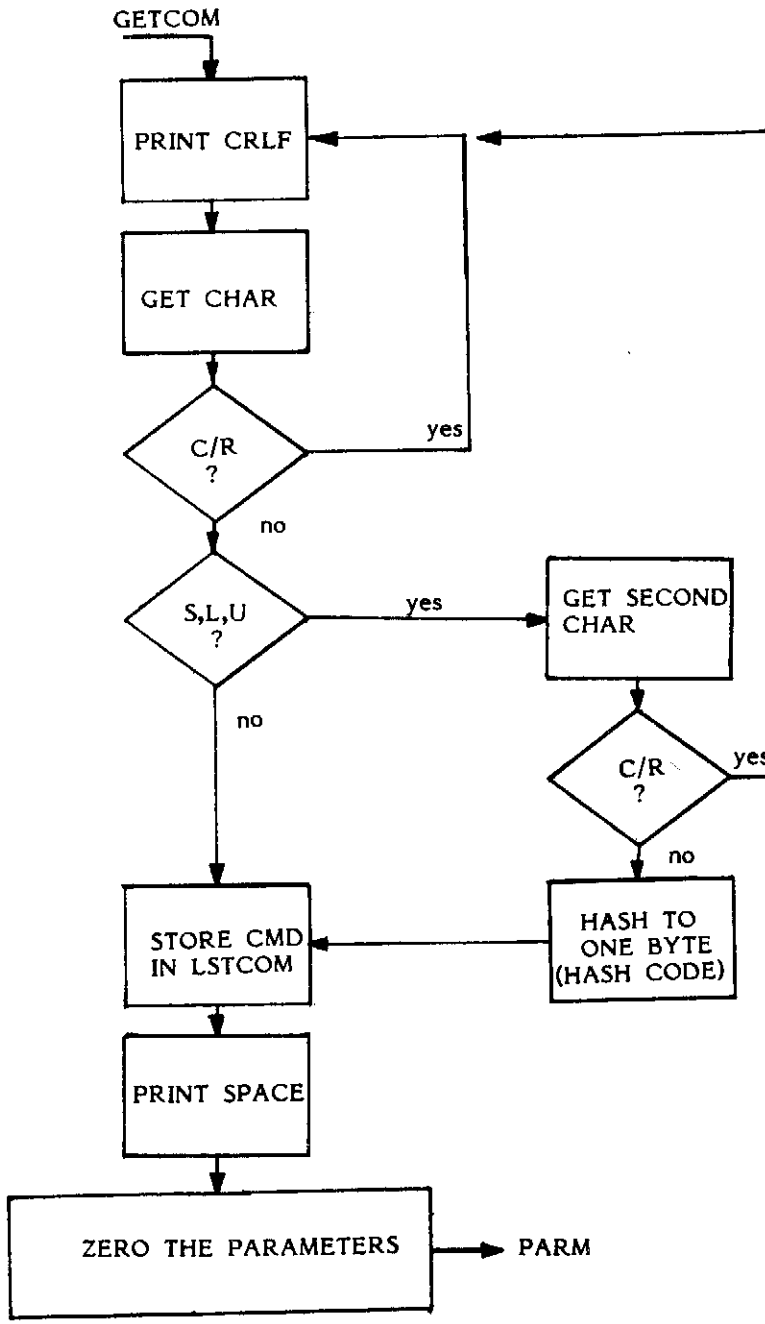


Figure 9-2. GETCOM FLOWCHART

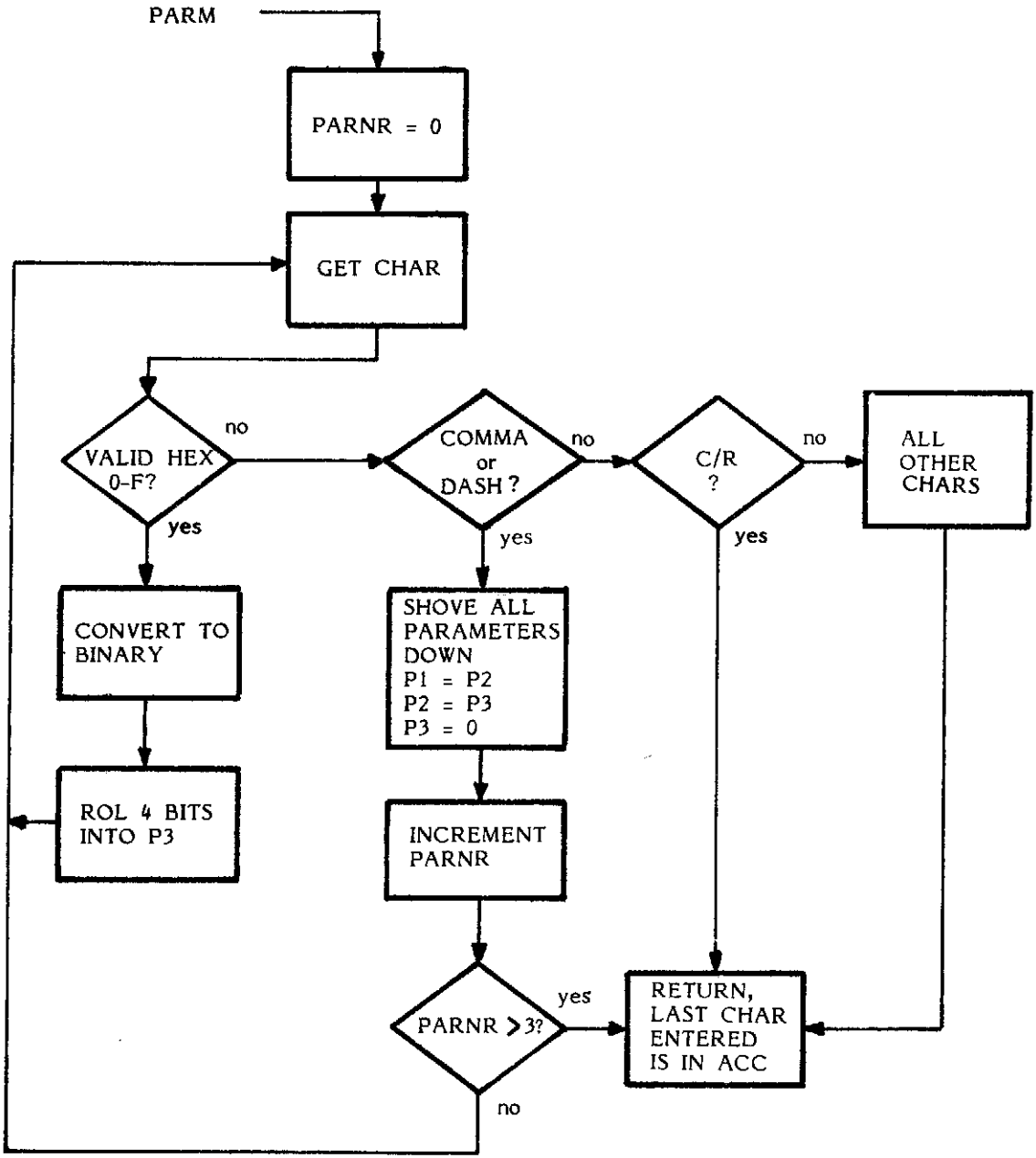


Figure 9-3. PARM FLOWCHART

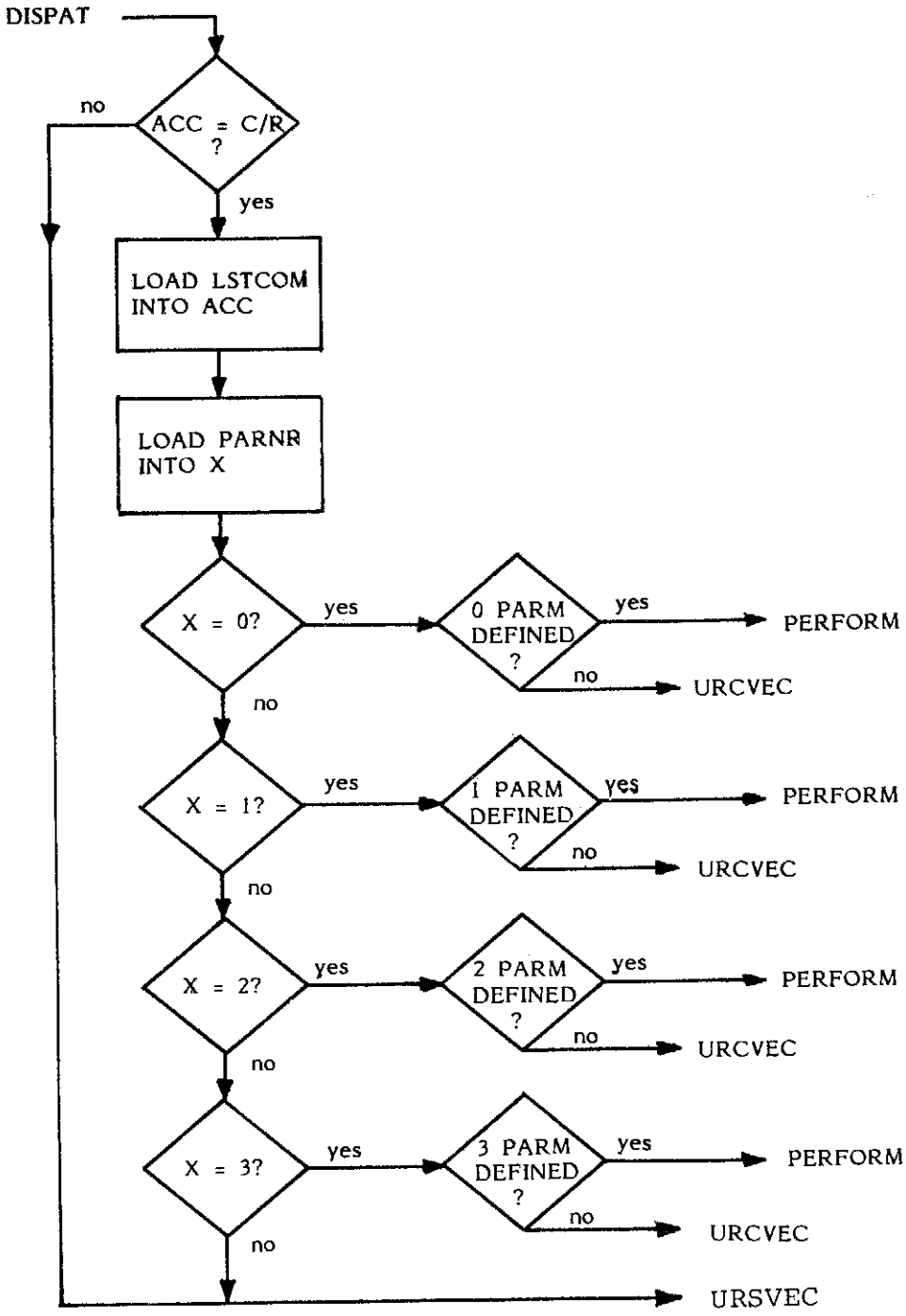


Figure 9-4. DISPAT FLOWCHART

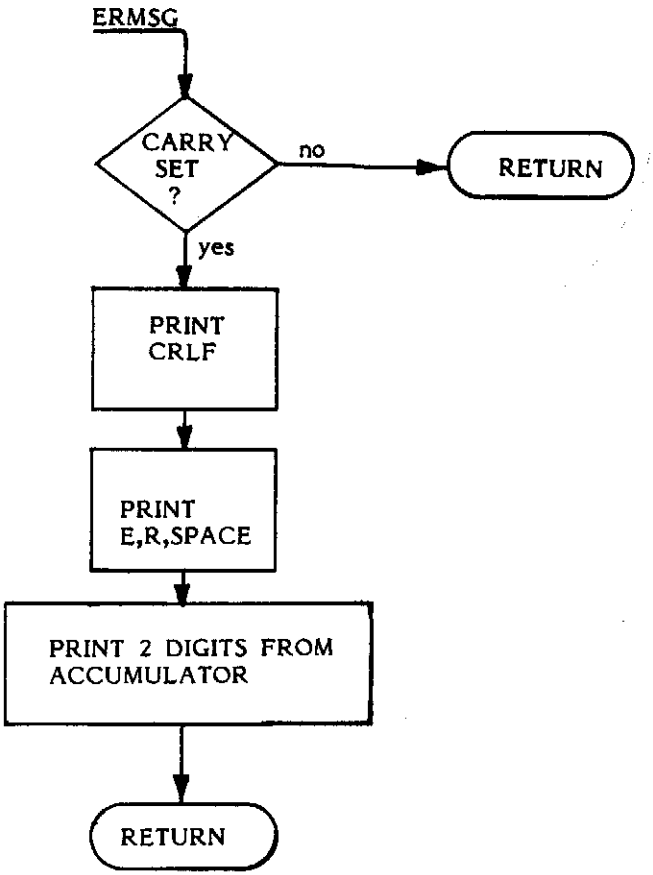


Figure 9-5. ERMSG FLOWCHART

9.4 VECTORS AND INTERRUPTS

A concept which is very important in understanding the SY6502 and SUPERMON is that of a transfer vector. A transfer vector consists of two or three locations at a fixed address in memory. These locations contain an address, or a Hex 4C **(JMP)** and an address. The address is in low-order, high-order byte order.

As an example, consider the function of outputting a character. In some cases, the character is to go to the display, in others to a terminal device. The action required in each case is radically different. It would be inefficient, in code and in time, to make the decision before outputting each character. The solution is a transfer vector. Whenever SUPERMON must output a character, it performs a JSR to OUTCHR. OUTCHR saves all registers, then performs a JSR to OUTVEC (at A663, in System RAM). If you are working at the Hex keyboard OUTVEC will contain a JMP HDOUT. HDOUT is the subroutine which will enter a character, in segment code, into the display buffer. If you are using a TTY or CRT, OUTVEC will contain a JMP TOUT. TOUT is the subroutine which sends a character, one bit at a time, to the serial I/O ports. When HDOUT or TOUT performs an RTS, control passes back to OUTCHR. OUTCHR restores the registers and performs an RTS, returning control to the caller.

Notice that the calling routine need not worry where the output is going. It is all taken care of by OUTCHR and OUTVEC.

When a vector is to be referenced by a JMP Indirect, only two bytes are required. Two-byte vectors are normally used only for interrupts.

An **INTERRUPT** is a method of transferring program control, or interrupting, the processor during execution. There are three interrupts defined on the SY6502:

NMI	--	non-maskable interrupt
RST	--	reset/power-on
IRQ	--	interrupt request

When one of these interrupts occurs, the processor pushes the PC register and the Flags register onto the stack, and gets a new PC from the **INTERRUPT VECTOR.** The interrupt vectors are located at the following addresses:

FFFA,FFFB	--	NMI
FFFC,FFFD	--	RESET
FFFE,FFFF	--	IRQ

These locations must contain the addresses of programs which will determine the cause of the interrupt, and respond appropriately.

In the SYM-1, System RAM (A600-A67F) is duplicated at FF80-FFFF (it is "echoed" there). On Reset, SUPERMON points these vectors to its own interrupt-handling routines. When an interrupt occurs, SUPERMON displays the address where the interrupt occurred with one of the following codes indicating the cause of the interrupt:

0	=	BRK instruction
1	=	IRQ
2	=	NMI
3	=	USER ENTRY (caused by JSR to USRENT at 8035)

Because all registers are saved, a **(G) (CR)** will cause execution to resume at the point of interruption. The user can intercept interrupt handling by inserting pointers to user interrupt routines in TRCVEC, UBRKVC, NMIVEC, or IRQVEC. See Section 9.8.2 for a discussion of the User Entry pseudo-interrupt. Table 9-2 describes all vectors used by the Monitor.

Table 9-2. SUPERMON VECTORS

NAME	LOCATION	FUNCTION
INVEC	A660-A662	Points to input driver.
OUTVEC	A663-A665	Points to output driver.
INSVEC	A666-A668	Points to routine which determines whether or not a key is down.
URCVEC	A66C-A66E	Unrecognized command. All unrecognized commands and parameter entry errors vectored here. Points to a sequence of: SEC - Set Carry RTS - Return
SCNVEC	A66F-A671	Points to routine which performs one scan of display from DISBUF.
EXEVEC	A672-A673	Points to RIN - get ASCII from RAM subroutine. The Execute (E) command temporarily replaces INVEC with EXEVEC, saving INVEC in SCRA, SCRB. The Hi byte of EXEVEC must be different from the Hi byte of INVEC.
TRCVEC	A674-A675	May be used to point to user trace routine after TRCOFF (See Section 9.6).
UBRKVC	A676-A677	May be used to point to user BRK routine after IRQVEC.
UIRQVC	A678-A679	May be used to point to user NON-BRK IRQ routine after IRQVEC.
NMIVEC	A67A-A67B	Points to routine which saves registers, determines whether or not to trace, based on TV.
IRQVEC	A67E-A67F	Points to routine which saves registers, determines whether or not BRK has occurred, and continues thru UBRKVC or UIRQVC.

9.5 DEBUG ON and TRACE

When the DEBUG ON key on your SYM-1 is depressed, DEBUG mode is established. In DEBUG mode, an NMI interrupt occurs every time an instruction is fetched from an address that is not within the monitor. SUPERMON's response is to save the registers and display the PC, with code 2 (for NMI). With each **(G) (CR)**, one instruction of the user program will be executed. This is called Single-Stepping.

In order to TRACE, alter the Trace Velocity (TV, at A656) to a non-zero value. (09 is a good value.) If you now enter **(G) (CR)**, SUPERMON will display the PC and the contents of the accumulator, pause, and resume execution. Addresses and accumulator contents will flash by one at a time. To stop the flow, depress any key (Hex keyboard) or the BREAK key (terminal). Execution will halt. A **(G) (CR)** will resume execution. The length of the delay is related to TV (not linearly; try different values) and, of course, the baud rate, if you are working from a terminal.

9.6 USER TRACE ROUTINES

As the complexity of your programs increases, you may wish to implement other types of trace routines. To demonstrate how this is done, an example of a user trace routine is provided in Figure 9-6. It prints the op code of the instruction about to be executed, instead of the accumulator contents.

But first of all, we don't want to be interrupted during trace mode by responding to an interrupt (a problem called recursion). SUPERMON will handle this by turning DEBUG OFF, then back ON. However, to implement this program control of DEBUG, you must add jumpers W24 and X25 to your SYM-1 board (see Chapter 4).

Now that you have added the jumpers, we are ready to enter the program UTRC and change vectors.

First, enter the program UTRC as given in Figure 9-6. Then change NMIVEC to point to TRCOFF, which will save registers, turn DEBUG OFF, and vector thru TRCVEC:

SD 80C0,A67A **(CR)**

Now, point TRCVEC to UTRC.

SD 0380,A674 **(CR)**

Enter a non-zero value in TV, depress DEBUG ON, and you're ready to trace.

NOTE: BRK instructions with DEBUG ON will operate as two-byte instructions and should be programmed as 00,EA (BRK,NOP).

Also, the first instruction after leaving SUPERMON will not be traced.

```
LINE #  LOC     CODE        LINE

0002    0000            ; UTRC - USER TRACE ROUTINE -
0003    0000            ; PRINT NEXT OP CODE INSTEAD OF ACCUMULATOR
0004    0000            ;
0005    0000            OPPCOM =$8337           ;PRINT PC, PRINT COMMA
0006    0000            PCLR   =$A659
0007    0000            PCHR   =$A65A
0008    0000            OBCRLF =$834A           ;PRINT BYTE FROM ACC, PRINT CRLF
0009    0000            DELAY  =$835A           ;DELAY BASED ON TV
0010    0000            WARM   =$8003           ;WARM MONITOR ENTRY
0011    0000            TRACON =$80CD           ;TURN TRACE ON, RESUME EXECUTION
0012    0000            TV     =$A656           ;TRACE VELOCITY
0013    0000            ;
0014    0000                   *=$380           ;PUT IN HI RAM (ENTIRELY RELOCATE/
0015    0380  20 37 83  UTRC   JSR OPPCOM       ;PRINT PC, COMMA
0016    0383  AD 59 A6         LDA PCLR         ;USE PC AS PTR TO OP CODE
0017    0386  85 F0            STA $F0
0018    0388  AD 5A A6         LDA PCHR
0019    038B  85 F1            STA $F1
0020    038D  A0 00            LDY #0
0021    038F  B1 F0            LDA ($F0),Y      ;PICK UP OP CODE
0022    0391  20 4A 83         JSR OBCRLF       ;OUTPUT OP CODE, CRLF
0023    0394  AE 56 A6         LDX TV           ;GET TRACE VELOCITY
0024    0397  F0 05            BEQ NOGO         ;NOGO IF ZERO
0025    0399  20 5A 83         JSR DELAY        ;DELAY ACCORDING TO TV
0026    039C  90 03            BCC GO           ;CARRY SET IF KEY DOWN
0027    039E  4C 03 80  NOGO   JMP WARM         ;HALT
0028    03A1  4C CD 80  GO     JMP TRACON       ;CONTINUE
0029    03A4                   .END
```

Figure 9-6. LISTING OF SAMPLE USER TRACE ROUTINE

USER TRACE EXAMPLE

```
.V 200,20A (CR)
  0200 A9 00 A9 11 A9 22 A9 33,0A
  0208  4C 00 02,58
0358
.SD 80C0,A67A (CR)          Vector modification
.SD 380,A674 (CR)           Vector modification
.G 200 (CR)                 Single-Step (Remember
0202,A9                     to set DEBUG ON before
                            each (G) (CR)
 G (CR)
0204,A9

.M A656(CR)
A656,00,09(CR)              Trace Velocity = 9
A657,4D (CR)
.G 200 (CR)
0202,A9
0204,A9
0206,A9
0208,4C                     Continuous trace of op codes
0200,A9
0202,A9
0204,A9
0206,A9
0208,4C
0200,A9
0202,A9
```

9.7 MIXED I/O CONFIGURATIONS

The Reset routine that is activated when power is turned on or RST is pressed establishes the terminal I/O configuration by loading a specified value into a location in System RAM, TOUTFL (A654). The high-order four bits of TOUTFL define which terminal devices may be used for input and output. A "1" signifies that a device is enabled, a "0" that it is disabled. The meaning of each bit and the values assigned at system reset are shown below. The routine referenced by entry (1) in the JUMP table will enable the TTY for input. For other configurations, load the appropriate value into TOUTFL.

TOUTFL bit:	7	6	5	4
default value:	1	0	1	1
meaning:	CRT INPUT	TTY INPUT	TTY OUTPUT	CRT OUTPUT

Bits 6 and 7 of another location in System RAM, TECHO (A653), are used to inhibit serial output (bit 6) and to control echo to a terminal (bit 7). Bit 6 may be toggled by entering "(CONTROL) O" (0F Hex) on the terminal keyboard or in software. The possible values for TECHO are shown below.

TECHO	80	echo output	(default value)
	C0	echo no output	
	40	no echo no output	
	00	no echo output	

With this information, you can alter the SUPERMON standard I/O configurations to suit your special needs. A common use would be routing your output to a terminal while using the Hex keyboard as an input device. Two possible ways of doing this will be discussed.

First, by merely altering SDBYT and OUTVEC, your input and echo will use the on-board keyboard and display, while Monitor and program output will go to the serial device. Choose the proper baud rate value for your device from the following table and put it in SDBYT (at A651) with the "M" command. Then enter the address of TOUT into OUTVEC from the hex keyboard as follows:

.SD 8AA0,A664 **(CR)**

Terminal Baud Rate	Value Placed in SDBYT
110	D5
300	4C
600	24
1200	10
2400	06
4800	01

Second, if you wish your input to be echoed on the terminal device, a small program must be entered. First, complete the sequence discussed above. Then, enter the following program:

```
UIN   JSR   GETKEY     20   AF   88
      BIT   TECHO      2C   53   A6
      BPL   UOUT       10   03
      JMP   OUTCHR     4C   47   8A
UOUT  RTS              60
```

Enter the program called "UIN" above at any user RAM location. Then use the "SD" command to put the address of UIN into INVEC (at A661) as follows:

.SD (UIN),A661 **(CR) (ENTER AT HKB)**

where (UIN) is the address of the program UIN.

9.8 USER MONITOR EXTENSIONS

Having read the section on Monitor flow, you will have noticed that unrecognized commands and parameter entries are vectored through URCVEC (A66C-A66E), which normally points to a SEC, RTS sequence at 81D1. By pointing URCVEC to a user-supplied routine in RAM or PROM, SUPERMON can easily be extended. The following example will illustrate the basic principle; many more sophisticated extensions are left to your imagination.

9.8.1 Monitor Extension Example

This example will define U0 with two parameters as a logical AND. The parameters and the result are in Hexadecimal.

```
LOGAND   CMP  #$14        ;USR0
         BNE  NEXT
         CPX  #2          ;two parms
         BNE  NEXT
DOAND    LDA  P2H
         AND  P3H         ;here's the 'and' hi
         TAX
         LDA  P2L
         AND  P3L         ;'and' lo
         JSR  CRLF        ;get new line
         JSR  SPACE
         JMP  OUTXAH      ;PRINT X and A
NEXT     SEC              ;last
         RTS
        .END
```

To attach LOGAND to the monitor, it must be assembled (probably by hand), entered into memory, and URCVEC altered to contain a JMP to LOGAND. Notice that more than one command could have been added, by pointing NEXT to the next possible command, instead of a RTS.

9.8.2 SUPERMON As Extension to User Routines

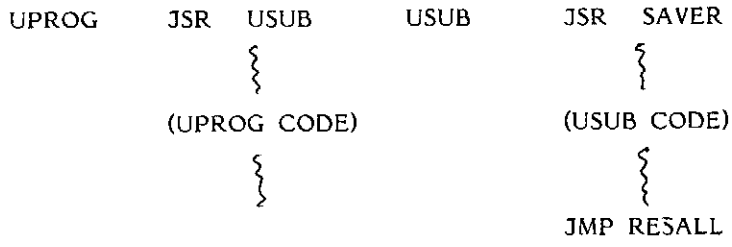
Because SUPERMON contains a user entry, it can easily be appended to your software. An example of the utility of this feature is a user trace routine, which could have an 'M' command, which would direct it to make SUPERMON available to the user. Here's what the code would look like.

```
UTRACE
. . .                                   Trace code

          JSR INCHR
          CMP #'M
          BNE ELSE
          JSR USRENT
          JMP UTRACE
          . . .

ELSE                                    Code executed if character
                                        input is not 'M'.
```

In this example, the user will type an 'M' to get into monitor, and a **(G) (CR)** to return to the calling portion of UTRACE. Note that the user PC and S registers should not be modified while in monitor if a return to UTRACE is intended.

9.9 USE OF SAVER AND RES ROUTINES

SAVER and the RES routines are designed to be used with subroutines. Their usage is as follows:



In this example, UPROG calls USUB. USUB calls SAVER, performs its function, and then jumps to RESALL. RESALL restores all registers and returns to UPROG. If RESXF or RESXAF were used instead of RESALL, UPROG would receive the F, or F and A registers as left by USUB.

APPENDIX A

IMMEDIATE ACTION

Your SYM-1 microcomputer has been thoroughly tested at the factory and carefully packed to prevent damage in shipping. It should provide you with years of trouble-free operation. If your unit does not respond properly when you attempt to apply power, enter commands from the keyboard, or attach peripheral devices to the system, do not immediately assume that it is defective. Re-read the appropriate sections of this manual and verify that all connections have been properly wired and all procedures properly executed.

If you finally conclude that your SYM-1 is defective, you should return it for repair to an authorized service representative. Specific instructions for obtaining a service authorization number and shipping your unit are contained with warranty information on the card entitled "LIMITED WARRANTY AND SERVICE PLAN" that is included with system reference material.

APPENDIX B

PARTS LIST

MATERIALS AND ACCESSORIES

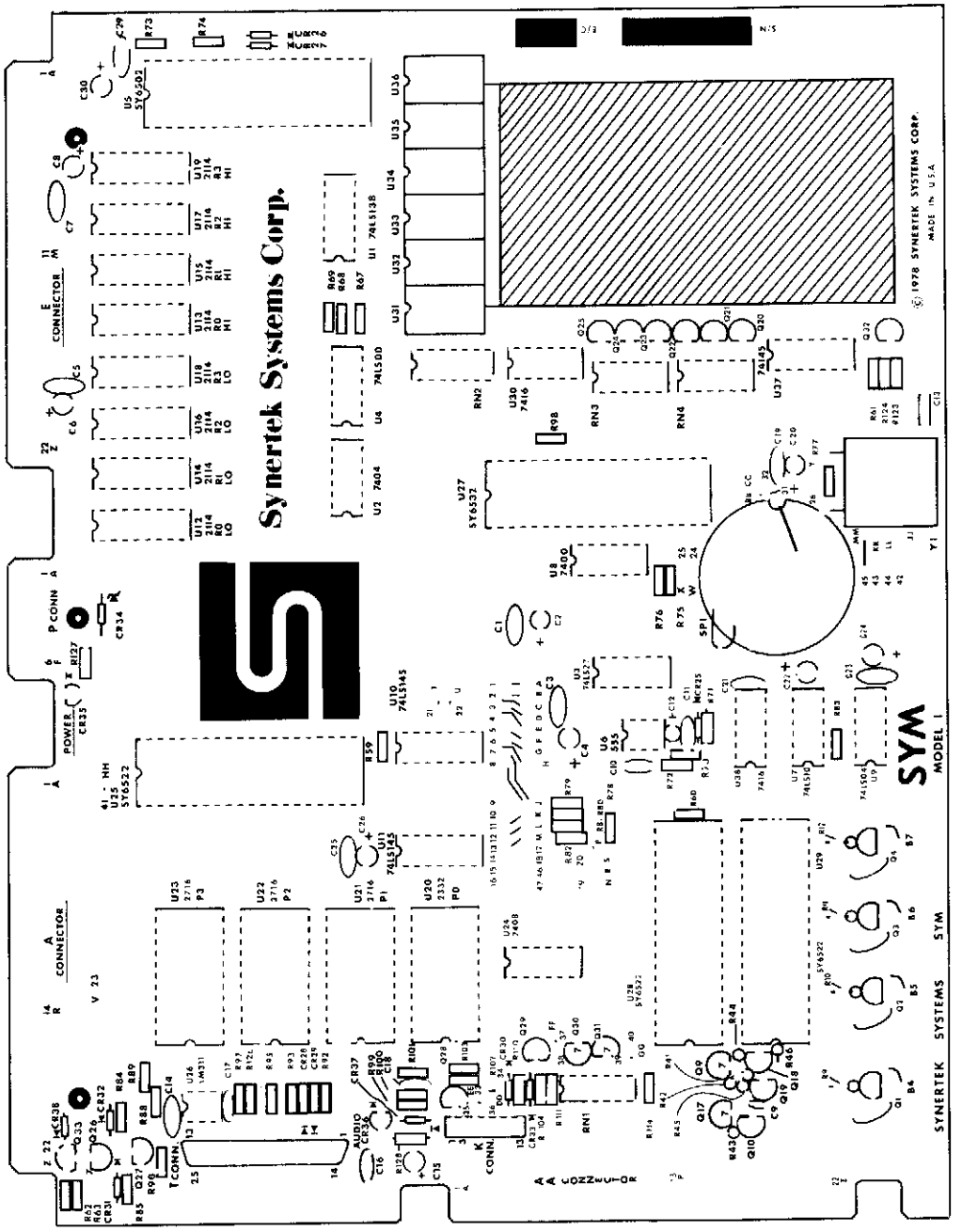
QTY.	DESCRIPTION	MANUFACTURER/PART NUMBER
1	CONNECTOR, DUAL 22/44	Microplastic 15622DPIS
1	CONNECTOR, DUAL 6/12	Teka TP3-061-E04
6	RUBBER FEET	3M SJ5018
1	SYNERTEK SOFTWARE MANUAL	MNA-2
1	SYM-1 REFERENCE CARD	SRC-1
1	SYM REFERENCE MANUAL	MNA-1
1	SYM-1 PC BOARD ASSEMBLY	
1	RED FILTER	

SYM-1 PC BOARD COMPONENTS

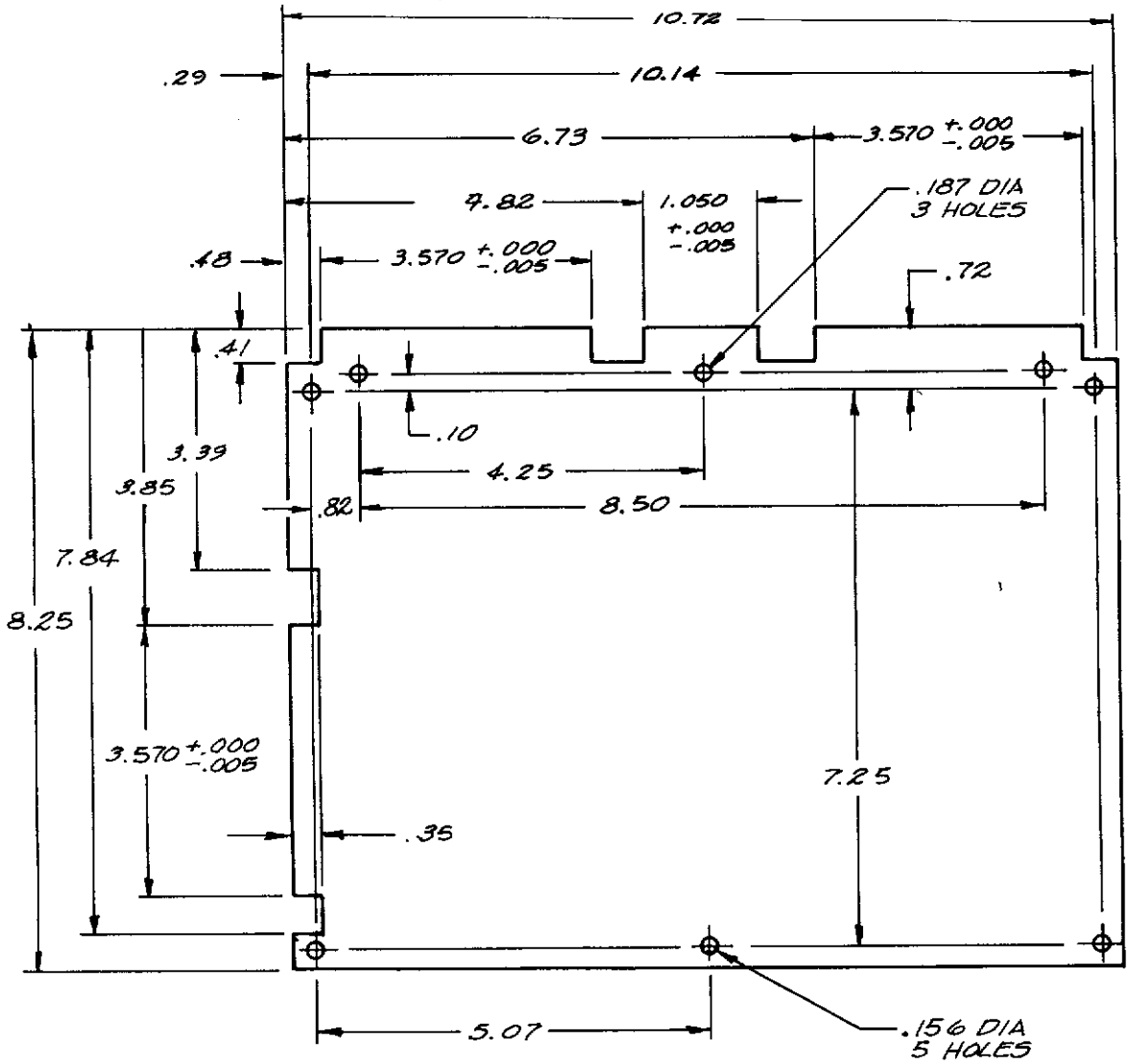
QTY.	DESCRIPTION	MFR. NO.	REFERENCE DESIGNATION
1	CPU	SYP6502	U5
2	VIA	SYP6522	U25,U29
1	RAM-I/O	SYP6532	U27
2	4K BIT RAM	SYP2114	U12, U13
1	32K BIT ROM	SYP2332	U20
1	NAND GATE	7400	U8
1	HEX INVERTER	7404	U2
1	AND GATE	7408	U24
2	HEX INVERTER-O.C.	7416	U30, U38
1	NAND GATE	74LS00	U4
1	HEX INVERTER	74LS04	U9
1	TRIPLE NOR GATE	74LS27	U3
1	TIMER	555	U6

QTY.	DESCRIPTION	MFR. NO.	REFERENCE DESIGNATION
1	DECODER	74LS138	U1
1	TRIPLE 3 INPUT NAND	74LS10	U7
1	DECODER	74145	U37
2	DECODER	74LS145	U10, U11
1	COMPARATOR	311	U26
1	RES-100 ohm, ¼W, 5%	RF14J100B	R128
3	RES-200 ohm, ¼W, 5%	RF14J200B	R43, 111, 114
1	RES-300 ohm, ¼W, 5%	RF14J300B	R107
4	RES-470 ohm, ¼W, 5%	RF14J470B	R84, 88, 124, 127
14	RES-1K, ¼W, 5%	RF14J1KB	R9-12, 41, 61-63, 73, 78, 85, 92, 97, 101, 113, 123
1	RES-1M, ¼W, 5%	RF14J1MB	R72
1	RES-2.2K, ¼W, 5%	RJ14J2.2KB	R103
14	RES-3.3K, ¼W, 5%	RF14J3.3KB	R42, 59, 60, 70, 74, 79-82, 87, 94, 95, 98, 126
10	RES-10K, ¼W, 5%	RF14J10KB	R45, 67-69, 75, 76, 83, 89, 93, 104
3	RES-47K, ¼W, 5%	RF14J47KB	R44, 46,71
1	RES-330K, ¼W, 5%	RF14J330KB	R77
2	RES-27K, ¼W, 5%	RF14J27KC	R90, 96
2	RES-150 ohm, ¼W, 5%	RF14J150B	R99, 110
1	RES-6.8K, ¼W, 5%	RF14J6.8KB	R100
1	CAP-10pf	DM15100J	C13
13	CAP - .01 mfd, 100V	DB203YZ1032	C1, 3, 5, 7, 10, 11, 17, 19, 21, 23, 25, 29
10	CAP - 10 mfd, 25V	T368B106K025PS	C2, 4, 6, 8, 12, 20, 22, 24, 26, 30

QTY.	DESCRIPTION	MFR. NO.	REFERENCE DESIGNATION
3	CAP - .1 mfd, 50V	3429-050E-104M	C9, 18
1	CAP - .22mfd,		C16
2	CAP - .47 mfd	C330C474M5V5EA	C15
1	CAP - .0047 mfd	UR2025100X7R472K	C14
12	NPN TRANSISTOR	2N2222A	Q1-4, 10, 18, 19, 27-29, 32, 33
11	PNP TRANSISTOR	2N2907A	Q9, 17, 20-26, 30, 31
11	DIODE, G.P.	1N914	CR25-33, 37, 38
1	DIODE, ZENER	1N4735	CR34
4	SOCKET - 24-PIN DIP	TIC8424-02	SK20-23
5	SOCKET - 40-PIN DIP	TIC8440-02	SK5, 25, 27-29
8	SOCKET - 18-PIN DIP	TIC8418-02	SK12-19
1	KEYBOARD		KB1
1	PC BOARD		PC1
6	7-SEGMENT DISPLAY, 0.3"	MAN 71A	U31-36
2	LED	RL4850	CR35,36
1	SPEAKER	70057	SP1
1	CRYSTAL	CY1A	Y1
	TAPE - 1½" x 2" STRIP		
1	RES. PACK - 150 ohm	898-3-R150	RN2
1	RES. PACK - 3.3K ohm	899-3-R3.3K	RN1
2	RES. PACK - 1K ohm	899-3-R1K	RN3, RN4



COMPONENT LAYOUT



OUTLINE DRAWING

B-5

APPENDIX C

AUDIO TAPE FORMATS

HIGH-SPEED FORMAT -- High speed data transfer takes place at 185 bytes per second. Every byte consists of a start bit (0), followed by eight data bits. The least significant bit is transmitted first. A "1" bit is represented by 1 cycle of 1400 Hz, while a "0" bit is represented by ½ cycles of 700 Hz. Physical record format is shown below.

8 sec. "mark"	256 SYN chars.	*	ID	SAL	SAH	EAL +1	EAH +1	DATA	/	CKL	CKH	EOT	EOT

8 sec. "mark" - Allows the tape to advance beyond the leader and creates an inter-record gap.

SYN (16 Hex) - ASCII synch characters that allow the SYM-1 to synchronize with the data stream.

* (2A Hex) - ASCII character that indicates the start of a valid record.

ID - Single byte that uniquely identifies the record.

SAL - Low order byte of the Starting Address from which data was taken from memory.

SAH - High order byte of the Starting Address from which data was taken from memory.

EAL +1 - Low order byte of the address following the Ending Address from which data was taken from memory.

EAH +1 - High order byte of the address following the Ending Address from which data was taken from memory.

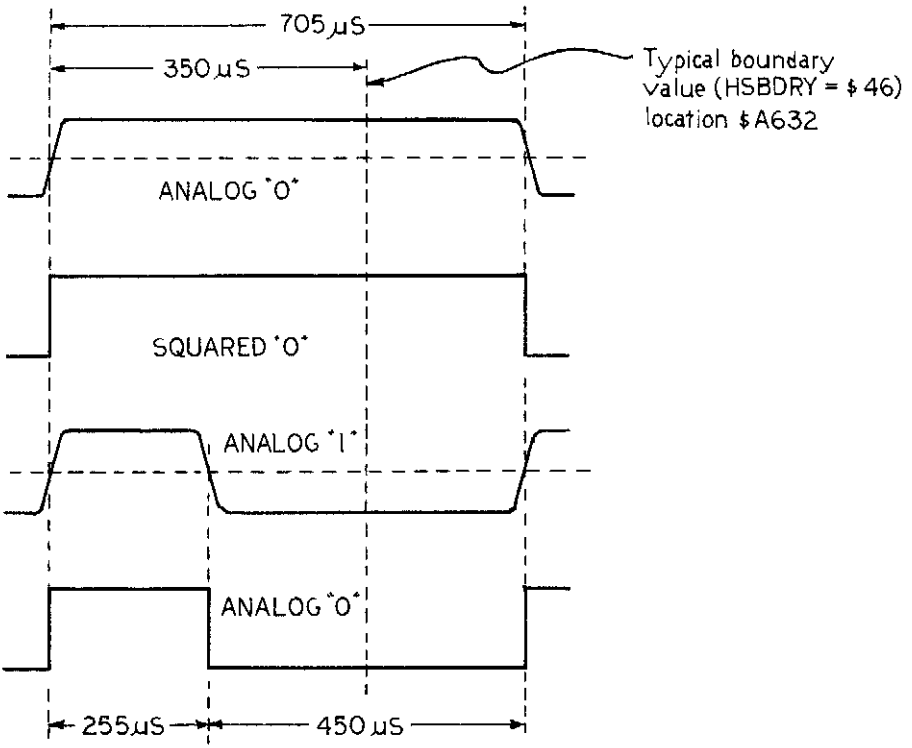
DATA - Data bytes.

/ (2F Hex) - ASCII character that indicates the end of the data position of a record.

CKL - Low order byte of a computed checksum.

CKH - High order byte of a computed checksum.

EOT (04 Hex) - ASCII characters that indicate the end of the tape record.



NAME	LOCATION	DEFAULT VALUE	DESCRIPTION
HSBDRY	A632	$46 (350µS)	HS Boundary
TAPET1	A635	$33 (255µS)	HS First Half "1" Bit
TAPET2	A63C	$5A (450µS)	HS Second Half "1" Bit

HIGH SPEED AUDIO FORMAT BIT WAVEFORMS

KIM FORMAT -- Data transfer in KIM format takes place at approximately 8 bytes per second. A "1" bit is represented by 9 cycles of 3600 Hz followed by 18 cycles of 2400 Hz, while a "0" bit is represented by 18 cycles of 3600 Hz followed by 6 cycles of 2400 Hz. Each 8-bit byte from memory is represented by two ASCII characters. The byte is separated into two half-bytes, then each half-byte is converted into an ASCII character that represents a Hex digit. The least significant bit is transmitted first. The KIM physical record format is shown below.

128 SYN chars.	*	ID	SAL	SAH	DATA	/	CKL	CKH	EOT	EOT

The sync characters, the ASCII characters "*" (2A Hex) and "/" (2F Hex) as well as ID, SAL, SAH, CKL, CKH and EOT serve the same functions as in HIGH-SPEED format. Sync characters, *, / and EOT are represented by single ASCII characters, while the remaining record items require two ASCII characters. Note that EAL and EAH are not used in the KIM format.

APPENDIX D

PAPER TAPE FORMAT

When data from memory is stored on paper tape, each 8-bit byte is separated into two half-bytes, then each half-byte is converted into an ASCII character that represents a Hex digit (O-F). Consequently, two ASCII characters are used to represent one byte of data. In the paper tape record format shown below, each N, A, D, and X represents one ASCII character.

; N_1N_0 $A_3A_2A_1A_0$ $(D_1D_0)_1$ $(D_1D_0)_2$ . . . $(D_1D_0)_n$ $X_3X_2X_1X_0$

;	- Start of record mark
N_1N_0	- Number of data bytes in (Hex) contained in the record
$A_3A_2A_1A_0$	- Starting address from which data was taken
(D_1D_0)-$(D_1D_0)_n$	- Data
$X_3X_2X_1X_0$	- 16-bit checksum of all preceding bytes in the record including N_1N_0 and $A_3A_2A_1A_0$, but excluding the start of record mark.

A single record will normally contain a maximum of 16 (10 Hex) data bytes. This is the system default value that is stored in system RAM at power-up or reset in location MAXRC (A658). You can substitute your own value by storing different number in MAXRC. To place an end of file after the last data record saved, place the TTY in local mode punch on, and enter ;00 followed by **(CR).**

APPENDIX E

SYM COMPATABILITY WITH KIM PRODUCTS

If you are a SYM-1 user who has peripheral devices which you have previously used with the KIM system or software which has been run on a KIM module, you'll find SYM to be generally upward compatible with your hardware and software. The following two sections describe the levels of compatability between the two systems to allow you to undertake any necessary modifications.

E.1 HARDWARE COMPATABILITY

Table E-1 describes the upward compatability between SYM and KIM at the Expansion (E) connector, while Table E-2 describes the compatability on the Applications (A) connector.

I/O port addresses differ between the two systems; you should consult the Memory Map in Figure 4-10 for details.

Power Supply inputs are provided on a separate connector with SYM-1, which means that if you have been using your power supply with a KIM device it will be necessary to rewire its connections to use the special connector on the SYM-1 board.

E.2 SOFTWARE COMPATABILITY

Table E-3 lists important user-available addresses and routines in the KIM-1 monitor program and their counterparts in SYM-1's SUPERMON. Most of the routines do not perform identically in the two systems. Before using them, check their operation in Table 9-1.

Table E-1. EXPANSION CONNECTOR (E) COMPATABILITY

SYM DESCRIPTION	SYM NAME	PIN #	KIM NAME	KIM DESCRIPTION
Jumper (Y,26) Selectable: OFF - Open Pin ON - Debug On/Off Output (U8-8)	DBOUT	17	SSTOUT	From (SYNC o NOT MONITOR) U26-6
Power On Reset Signal Output: "0" After power on "1" When reset by software	$\overline{POR}$	18		No equivalent

Table E-2. APPLICATION CONNECTOR (A) COMPATABILITY

SYM DESCRIPTION	SYM NAME	PIN #	KIM NAME	KIM DESCRIPTION
Jumper (V,23) Selectable: OFF - Open Pin ON - Remote Audio Control Out	AUD.RC	N	+12V	+12V Not required on SYM
Jumper (HH,41) Selectable: OFF Open Pin ON $\overline{\text{ICXX}}$ Decode Out		K	DECODE Enable	Enable 8K Decoder

Table E-3. SYM-KIM SOFTWARE COMPATABILITY

SYM		KIM		FUNCTION
Label	Address(es)	Label	Address(es)	
PCLR	A659	PCL	00EF	Program Counter - low
PCHR	A65A	PCH	00F0	Program Counter - high
FR	A65C	PREG	00F1	Status Register
SR	A65B	SPUSER	00F2	Stack Pointer
AR	A65D	ACC	00F3	Accumulator
YR	A65F	YREG	00F4	Y - Register
XR	A65E	XREG	00F5	X - Register
SCR6	A636	CHKHI	00F6	Checksum - low
SCR7	A637	CHKSUM	00F7	Checksum - high
P2L	A64C	SAL	17F5	Start Addr Low - audio/paper tape
P2H	A64D	SAH	17F6	Start Addr High - audio/paper tape
P3L	A64A	EAL	17F7	End Addr+1 Low - audio/paper tape
P3H	A64B	EAH	17F8	End Addr+1 High - audio/paper tape
P1L	A64E	ID	17F9	ID Byte audio Tape
NMIVEC	A67A-B FFFA-B	NMIV	17FA-B FFFA-B	NMI Vector
RSTVEC	FFFC-D	RSTV	17FC-D FFFC-D	Reset Vector
IRQVEC	A67E-F FFFE-F	IRQV	17FE-F FFFE-F	IRQ Vector
DUMPT	8E87	DUMPT	1800	Dump memory to audio tape
LOADT	8C78	LOADT	1873	Load memory from audio tape
CHKT	8E78	CHKT	194C	Compute checksum for audio tape
OUTBTC	8F4A	OUTBTC	195E	Output one KIM byte
HEXOUT	8F52	HEXOUT	196F	Convert LSD of A to ASCII AND write to audio tape

Table E-3. SYM-KIM SOFTWARE COMPATABILITY (Continued)

SYM		KIM		FUNCTION
Label	Address(es)	Label	Address(es)	
OUTCHT	8F5D	OUTCHT	197A	Write one ASCII character to audio tape
RDBYT	8E2C	RDBYT	19F3	Read one byte from audio tape
PACKT	8E3E	PACKT	1A00	Pack ASCII to nibble
RDCHT	8E61	RDCHT	1A24	Read one character from audio tape
RDBITK	8E0F	RDBIT	1A41	Read one bit from tape
SVNMI	809B	SAVE	1C00	Monitor NMI entry
RESET	8B4A	RST	1C22	Monitor RESET entry
OUTPC	82EE	PCCMD	1CDC	Display PC
INCHR	8A1B	READ	1C6A	Get character
LP2B+7	841E	LOAD	1CE7	Load paper tape
SP2B+4	869C	DUMP	1D42	Save paper tape
OUTS2	8319	PRTPNT	1E1E	Print pointer
OUTBYT	82FA	PRTBYT	1E3B	Print 1 byte as 2 ASCII character
INCHR	8A1B	GETCH	1E5A	Get character
DLYF	8AE6	DELAY	1ED4	Delay 1 bit time
DLYH	8AE9	DEHALF	1EEB	Delay ½ bit time
INSTAT	8386	AK	1EFE	Determine if key is down
OUTDSP	89C1	SCAND	1F19	Output to LED display
SCAND	8906	SCANDS	1F1F	Scan LED display
INCCMP	82B2	INCPT	1F63	Increment pointer
GETKEY	88AF	GETKEY	1F6A	Get key
CHKSAD	82DD	CHK	1F91	Compute checksum
INBYTE	81D9	GETBYT	1F9D	Get 2 Hex characters and pack

APPENDIX F

CREATING AND USING A SYNC TAPE

To read serial data from tape, the SYM-1 makes use of a stream of SYNC characters which form the leader of every tape record. For a complete description of audio tape record formats, refer to Appendix C.

The audio signal appears on the T and A connectors in two forms: Audio Out (HI) and Audio Out (LO). The only difference between these signals is their magnitude. It is usually best to connect Audio Out (LO) to the MIC input of your recorder.

When the SYM-1 searches for a record, an 'S' and a decimal point are displayed until the SYNC characters are recognized. However, if the volume and tone controls on the recorder are not set correctly, the SYNC characters will not be recognized, the 'S' on the display will not go out, and no data will be loaded into memory.

Before attempting to save and load data for the first time, it will be helpful to generate a SYNC tape to use for adjusting the controls.

To generate a SYNC tape, modify memory location TAPDEL, A630 to FF:

```
.M A630 )
A630, 04, FF
A631, 2C, )
.
```

Place a blank tape in the recorder, depress RECORD and PLAY on the recorder, and enter a SAVE command:

```
.S2 200-201 )
```

The length of the tape leader, determined by TAPDEL will be over 6 minutes.

When the recording is finished (the display re-lights), rewind the tape. Enter the load command:

```
.L2 )
```

The 'S' and decimal point should light. Start the tape with the PLAY button on the recorder and adjust the volume and tone controls on the recorder until the 'S' goes out and stays out.

You can now remove the SYNC tape and proceed to save and load actual programs and data.

APPENDIX G

MONITOR ADDENDA

1. The DBOFF routine does not debounce the DEBUG-ON switch; therefore, user programs should not be interrupted by depressing DEBUG-ON while using a user trace routine or while OUTVEC points to a user routine. (This will cause recursive interrupts.)

2. The audio cassette software will not read or write location $FFFF. Use $A67F ($A600 thru $A67F is echoed at $FF80 thru $FFFF).

3. The DEBUG-ON switch bounces, therefore it should not be used to interrupt user programs while using a user trace routine or while OUTVEC points to a user routine. (This will cause recursive interrupts.)

APPENDIX H

SUPPLEMENTARY INFORMATION

Changing Automatic Log-On

After power is applied to the SYM, SUPERMON waits for the keyboard or the device connected to PB7 on the 6532 (normally the RS232 device) to become active. PB6 (the current loop device) is ignored because a disconnected current loop always looks active.

If you expect always to log-on a current-loop device, the following jumper change will eliminate the necessity of entering (SHIFT) (JUMP) (1):

Change CC-32 and BB-31
to CC-31 and BB-32

Now the log-on for your current loop device is simply a "Q", entered at the device. (Note that you cannot now log-on automatically to the keyboard unless the current loop device is connected, and powered-up.)

Using On-Board LED Display

Because of the extensive use of transfer vectors in SUPERMON, the same monitor calls can be used to activate the LED display as for terminal devices. The major difference is that you must call ACCESS (address 8B86) before outputting the first character in order to remove write-protection from the display buffer (DISBUF, address A640 thru A645).

If the SYM-1 was logged-on to from the HKB, each call to OUTCHR (address 8A47) will examine the ASCII character in the Accumulator, look up its segment code, shift everything in the display buffer of segment codes left one digit, place the new code in the rightmost digit, and scan the display once.

If the SYM-1 was logged-on to the HKB, each call to INCHR (address 8A1B) will scan the display from the codes in DISBUF continuously until a key is depressed (2 keys in the case of SHIFT keys, 4 in the case of SHIFT ASCII keys). The key will be fully debounced, the beeper beeped, the ASCII or HASHED ASCII code taken from a table, and passed back to the caller in the Accumulator. The Flags will reflect a compare with carriage-return.

Other useful routines are:

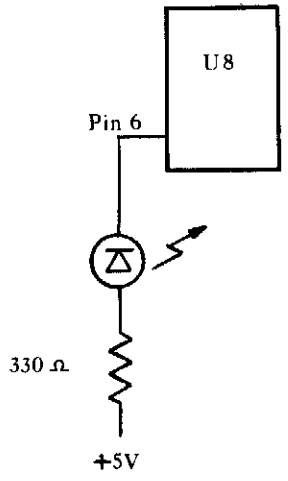
GETKEY (88AF)	Same as description of INCHR above, but disregard log-on and no compare performed.
OUTDSP (89C1)	Same as description of OUTCHR above, but disregard log-on.
KEYQ (8923)	Test for key depressed on HKB. On return, Z Flag = 1 if key down.
SCAND (8906)	Scan display once from segment codes in DISBUF. On return, Flags reflect call to KEYQ.

INSTAT (8386)	If logged-on to HKB, check for key down (else check for BREAK key). On return, carry set if key down (or BREAK key). Leading edge of key debounced.

See also chapter 9 for discussion of monitor calls.

Adding DEBUG Indicator

While using trace routines which turn DEBUG on and off, it is often desirable to have an external indication of the DEBUG state. The addition of an LED and a resistor as follows will achieve this.



U8 is a 14 pin package located above the beeper.

The LED will remain on while DEBUG is on.