SYM-1

MONITOR

THEORY OF OPERATIONS

MANUAL

By Robert A. Peck

Copyright 1980
Robert A. Peck
All Rights Reserved

Sym-1 is a trademark
Of

Synerteck Systems Corporation

INTRODUCTION-

This Monitor Theory of Operations Manual is intended for use by present owners of the SYM-1 Reference Manual. The primary purpose is to provide the user with a series of flow-narratives describing the sequence of events for most of the monitor routines.

The Monitor assembly language coding is, for the most part, reasonably well-commented. However due to the considerable effort at reuse of various routines, (and well done I might add) the flow and purpose of certain of the subroutines is not exactly obvious.

This manual then is intended to help the user who has some familiarity with assembly language programming to understand these routines more easily. By this means, these routines may find more use within the users programs.

One might have tackled this project by setting up a set of flow charts, one for each routine or subroutine. I felt that the flow-narrative approach would provide a reasonable amount of information as to intents and cases and would save the user the cost of the drawings and the magnitude of paper this would have taken.

If you should encounter difficulties in understanding any of these routines or discover any typos or context errors, I would appreciate hearing about them. The appropriate corrections/clarifications may then be added to the next printing, benefitting all other SYM-1 users. If you desire a written response to your inquiries, please include a self addressed stamped envelope. For more data, see notes in the Bibliography.

Robert A. Peck, May 1980.

ACKNOWLEDGEMENTS-

I wish to thank Synertek Systems for their support of this project, which included the permission to use excerpts of the Monitor listing in this manual. I would also like to thank Dr. H. Luxenberg for his support and encouragement.

General Introduction

Throughout the monitor routines, there is a need to store registers, access input and output ports, set up time intervals and to interact with the user. Since these are repetitive actions, various monitor subroutines have been defined which are used many times, rather than rewriting the same code each time. By this method, the monitor ROM is able to hold a much larger number of routines than would have been possible if no subroutine calls were used.

Use of these subroutines could probably reduce the amount of code which we have to write into our own programs. Therefore, the following sections attempt to explain the function and operating sequence of all of the monitor subroutines.

Handling the System RAM-

Various monitor routines require access to the system RAM. This RAM area is normally write-protected. When a monitor subroutine which writes the system RAM is called by another monitor subroutine, a JSR to the ACCESS subroutine always preceeds it. Therefore, to use these particular routines, we must either call ACCESS first, or modify the hardware to remove the write-protection from the system RAM. For any monitor routine which uses the SYSRAM, we must remember to call ACCESS before using that routine or we will get unpredictable results.

ACCESS - Call by 20 86 8B

Removes the write-protection from the system RAM. Has no effect on any registers because it uses SAVER subroutine first, then restores all registers afterwards. Sets U29 pin 2 to a logical 1. If the user has a need to speed up the routine, the same effect can be had by the following sequence: it is written as a subroutine and could be used as a substitute for ACCESS.

```
48          PHA         Save A
AD 01 AC    LDA  OR3A   Set this bit
09 01       ORA #01     to allow
8D 01 AC    STA OR3A    writing SYSRAM
AD 03 AC    LDA DDR3A   Define this
09 01       ORA #01     bit as an
8D 03 AC    STA DDR3A   output
68          PLA         Restore A
60          RTS         Return from subroutine
```

NACCES - Call by 20 9C 8B

Adds write-protection to system RAM. Has no effect on registers. Sets U29 pin 2 to logic 0. As with the ACCESS routine, it calls SAVER which stores all registers, then the RESALL routine restores all registers. Since the only register affected by the routine is the A accumulator, we could simulate this routine in fewer bytes by the following subroutine:

```
48          PHA           Save A
AD 01 AC    LDA  OR3A     Set this bit
29 FE       AND  #FE      to prevent
8D 01 AC    STA  OR3A     writing SYSRAM
AD 03 AC    LDA  DDR3A    Define this
09 01       ORA  #01      bit as an
8D 03 AC    STA  DDR3A    output
68          PLA           Restore A
60          RTS           Return
```

The purpose for presenting alternate ways of accomplishing certain of the functions is for illustration only. It is not intended to be a criticism of the methods actually used. It is intended rather to demonstrate, by example, what the routine is trying to accomplish. The general purpose usage of various subroutines such as SAVER, for example, made it much more expedient to use existing save-routines rather than to provide a special coding noted above for a single purpose. All such suggested alternatives which follow, then, are intended for a similar purpose; that is, as an aid to the users understanding of the monitor as written.

Register Save and Restore-

The previous example dealt with a substitute for the routine ACCESS Since the routine is used quite extensively throughout the monitor, it would seem an appropriate time to study this routine in depth.

The saver routine makes extensive use of the system stack. To be sure the reader is familiar with stack operation, a small review of the stack definition is in order.

The stack pointer is an internal register which points to the next available memory location in a RAM area where a data variable may be stored. Pushing an item onto the stack comprises storing the variable in the location specified by the stack pointer, then automatically decrementing the stack pointer to point to the next lower RAM location. Pulling an item off of the stack entails auto-incrementing the stack pointer, then moving the data from the location it now points to, into the selected register. Multiple "pushes" will stack the data into the memory on a last-in-first-out basis. As a final note, the stack pointer is only eight bits wide. There is an extra "01" in hex as a wired-in part of each stack address. Therefore, the stack points to an area 0100-01FF.

With this as background, we are about to proceed on the description of SAVER's operation. The chart below will illustrate in column one the value of the stack pointer. S represents the stack value just before entering the subroutine SAVER. S-1, S-2, etc represent intermediate values of the stack. During the SAVER routine, the stack pointer value is transferred to the index register. Column 2 represents the indexed location of the data. Column 3 represents the appearance of the stack after the registers are pushed in. Column 4 represents the final appearance of the stack after the data has been moved around a bit.

The reason the data is moved is to allow for three separate restore register routines after SAVER, specifically RESXAF, RESXF, and RESALL. The explanation for these follows this table.

Stack Value	Indexed Stack Data Location	Initial Contents	Final Contents	Comments-(final)
S	0109,X	PCH	Flags	Condition Codes
S-1	0108,X	PCL	A	Accumulator

S-2	0107,X	Flags	X	Index-X
S-3	0106,X	A	Y	Index-Y
S-4	0105,X	A	PCH	Return Address-
S-5	0104,X	A	PCL	from SAVER
S-6	0103,X	Flags		
S-7	0102,X	A		
S-8	0101,X	X		
S-9	0100,X	--		Stack value during data reshuffle

The stack was transferred to the index register so that we could precisely locate and reshuffle the stack contents. When we entered the SAVER routine, the return address from the routine was already on the stack at locations S and S-1. If we want to leave the registers on the stack when returning from SAVER, we have to move the return address down on the stack to a point beyond where the registers will be stored. Thus the reshuffle is needed.

For the sake of example, let's assume the stack value S was FF when we started to enter SAVER. It could be any value, but this will serve as a discussion point. Therefore, when all registers have been pushed onto the stack (after location 8191 has been executed), the stack actual value and now the index register current value will be F6.

The instruction at 8192 says LDA 0109,X. This means take the X register contents, add 109 hex, and use this as the effective address from which the A register will be loaded. In this case, the resultant address is 109 + F6 = 01FF. This means we are moving the PCH value to the accumulator (PCH = high byte of subroutine return address).

The rest of the address pointers into the stack area are viewed in the same manner.

No matter what the current value of the stack pointer happens to be, the SAVER routine will still function the same way. After 8191 is executed, the stack value is, in symbolic notation, "S-9" relative to its contents before SAVER is called. Therefore, to point to PCH value as shown above, we take (S-9) + 0100 + 9 which gives us the effective address where the PCH is stored.

When the RTS from SAVER is executed, the return address is pulled from the stack locations S-5, S-4 and becomes the new contents of the program counter. The stack value is now S-4 (4 less than before we entered the SAVER routine), since the now we have the saved flags, A, X and Y occupying positions within the stack.

To restore the registers, we have a choice. To restore all registers, we jump to RESALL (not JSR). This one increments the current stack value from S-4 to S-3, then moves this value into accumulator A, then from A into Y (Restores the original Y value). It restores X from location S-2 in the same manner. It then pulls A from location S-1 and pulls the flags from location S.

The routine RESXF and RESXAF are each just earlier entry points to the same program sequence which leads into RESALL. If we enter at RESXF, the current value of the flags will be stored at the stack location which had held the saved value of the flags. Then RESALL will put the current value of the flags back into the flag register on completion of the routine. The same holds true for RESXAF; we store the current A contents in place of the saved A contents, then continue at RESXF, then RESALL.

To allow the reader to follow the coding of RESXF and RESXAF, the table below shows the stack contents and indexed relative addressing as was done for the SAVER routine earlier. Again the value S refers to the stack pointer value before SAVER was executed.

RESXF:

Stack Value	Indexed Stack Data Location	Current Contents	Final Contents (before continuing (with RESALL)	
S	0104,X	Saved Flags	New flags	
S-1	0103,X	Saved A (or new A)	Saved A (or new A)	
S-2	0102,X	Saved X	Saved X	
S-3	0101,X	Saved Y	Saved Y	
S-4	0100,X	New flags	--------	stack value entering RESALL

RESXAF:

Stack Value	Indexed Stack Data Location	Current Contents	Final Contents (before continuing (with RESXF)	
S	0105,X	Saved Flags	Saved flags	
S-1	0104,X	Saved A	New A	
S-2	0103,X	Saved X	Saved X	
S-3	0102,X	Saved Y	Saved Y	
S-4	0101,X	New A	--------	stack value entering RESXF

At the end of the RESALL routine is an RTS, a return from subroutine. When we enter a particular subroutine where we must make use of the registers within the subroutine, it is wise to use SAVER first. Therefore, before SAVER was executed, the return address for the subroutine which calls SAVER is already on the stack (at relative locations S + 1 and S + 2).

After all of the registers and flags have been restored then, the stack value will be S. Thus the RTS causes it to become S + 1 then S + 2 as the return address from the subroutine (which called SAVER) is pulled off the stack. This restores the stack pointer to the same value it started from before the subroutine sequence began.

Hex Keypad I/O Routines:

The basic communications with the user of this single board computer takes place through the Hex Keypad and display (if no other I/O is available).

These keypad/display routines are described next.

The Hex Keyboard I/O routine takes care of not only the keyboard, but also scans the display. Since it serves as the main I/O technique for our single board computer, we should undestand how the routine functions. Many of the other routines use these I/O routines as well. The main KEYPAD I/O subroutine is GETKEY.

```
GETKEY  -  Call Saver: save register contents on the stack
           Call GK (explained next section) Returns with ASCII in A
           Was the key found the ASCII key? (hex "FE")
                   If not, exit the routine with the actual
                   ASCll or hash code in the A-register.
           If it as the ASCII key: Call GK, returns with hex in X
                                            transfer X to A
                                            shift A left 4 times
                                            store A in scratch loc. E
                                   Call GK, returns with hex in X
                                            transfer X to A
                                            clear carry
                                            add scratch loc. E to A
                                            return with ASCll in A
                   (Assembles a pair of hex digits into an ASCII
                   character equivalent)
           Restores all registers except A (where ASCII is) and F.
```

Subroutine GK, called by GETKEY, calls some subroutines of its own; and as we shall see, GK does most of the work in the hex keypad I/O routine.

GK - 1. Zeros out keyboard shift flag (so a shift can be sensed if it happens)

2. JSR To IJSCNV

This subroutine does a number of things, including scanning the keyboard and the display. It is a vectored scan routine in that IJSCNV points to location A670 in SYSRAM. IF we want an alternate output scan routine we must change the vector at A670 to point to our own scan subroutine; but we will for the moment continue talking about the default scan routine at 8906 as pointed to by the current IJSCNV vector. 8906 is the SCAND routine.

SCAND - Configures the 6532 ports as outputs. Port B to select which digit to be turned on (through U37 output) and Port A to output the segment codes from the DISBUF memory area for each digit as selected.

When entering SCAND, the A register is loaded with the value 9, then subroutine CONFIG is called.

CONFIG calls SAVER, then makes two loops through a routine which pulls values out of two tables. VALSP2 contains data direction register and I/O register control information for Port B of the 6532. Table VALS contains the same information for Port A.

The index register contents is the former register A contents and is used as the pointer into each of these tables. Thus, in this example, location VALS + 9 = 8BCF, so FF will be stored at location A401. The Y register is decremented by one (to zero) then the loop repeats, storing 00 at location A402 and 00 at A400. These actions have the effect of defining both Port A and Port B (except for most significant bit of Port A) as outputs and setting the initial value of all ouputs to zero.

SCAND then continues after configuration with two scan control loops. The outer loop consists of the selection of one of the 6 digits to display and the output of the selected digit's segment display data and of the DISBUF area to Port A. The inner loop is a timing loop at 891D-891F. This allows each digit to be active for a minimum of 80 clock cycles, then the next digit is selected.

To clarify this further, this is a multiplexed output display. This means that all six digits have all their segment a's connected together, all segment b's, etc. When we feed the data out to Port A, all six digits then receive the same command - to turn on specific segments. However, a path to the return side of the power supply is only provided for one single digit at a time through the action of the number output on Port B. Each digit is selected, turned on for a period of time, then turned off so the next one may be addressed. The scanning takes place fast enough and often enough that the user would believe that all digits are on simultaneously.

The routine proceeds as follows: Set up X-register as loop counter for six digits output. Load A register with the segment codes from (DISBUF + X) address. Set the output register to zero. (This prevents a ghost of any digit from appearing on the previous digit during the transition time). Turn on the ground return path for the next digit. Output the digit segment from register A. Load Y as a time delay loop counter. Stay in a loop for 80 clock cycles while the segments for that digit are being displayed. Repeat the process until all 6 digits have been scanned once.

Subroutine SCAND ends with an entry to KEYQ. This KEYQ routine calls CONFIG to define Port B as an output and all but the MSB of A as an output. It then reads the input Port A, tests the value and sets the Z flag to 1 if no key is down. (Result of the Port A data read is zero).

Continuing with our discussion of subroutine GK, at location 88D7, we have a BEQ GK1. As noted above, after SCAND, the Z flag will be set if any key is down This means that before any key is held down, the scanning of the keypad will continue while it is waiting for a keypress. When it senses a key down, the subroutine LRNKEY is executed. This stores what was read from Port A on the 6532 in scratch location F (after stripping off the two most significant bits). Then sets Port A as an output and Port B as an input on bits 0-3 so that it can determine which column of the keyboard has a key down. The information stored in location A63F (SCRF) defines which keyboard row was pressed and the data read from the input Port B defines which column. Therefore, the intersection of row and column defines the specific key pressed.

The routine LRNKEY concludes by determining in which column the key-press occurred and appending an ID number to the row information obtained from Port A. The number thus formed becomes the value in the accumulator against which the various entries in the ASCII to SYM table may be compared. The number in the accumulator is compared, one step at a time to those numbers in the table titled "SYM" (location 8BD6). The index register is incremented each time a comparison is made. When a match is found, the contents of the index register is moved into accumulator A, then is modified by the contents of the keyboard shift flag. It is transferred back to the X register where it is used as a counter into the table titled SC11 (location 8BEF). The accumulator is loaded from this table and a return is made to the calling routine (in this case) to 88DC.

After the keypress is interpreted, we push the ASCII data onto the stack and save the hex data from the x-register on the stack as well. Then call BEEP to tell the user a keypress has been sensed.

Now KEYQ is called again to see if the key is still down. If so, it stays in the loop calling KEYQ waiting for the release. (Try any key entry, as long as key is pressed, display stays blank waiting for release - operates during monitor routine execution). If key release is sensed, we use JSR NOBEEP as a delay tactic, then use KEYQ again to loop back if there was keybounce on release. At the exit, ASCII value is restored to A after the hex value is restored to X (if key was 0-9 or A-F). During the second part of the GETKEY routine, if the ASCII key had been sensed as the initial keypress, the routine pulls the hex value of the next keypress out of the X register, shifts it left 4 places and stores it in SCRE (scratch location E). Another pass through GK picks up another hex digit in the X register which is added to the previous shifted digit, forming a two digit ASCll representation of a pair of hex digits separately entered. With this in the accumulator, the routine GETKEY then ends normally, restoring the registers.

A special note is in order here regarding the difference between GK and GETKEY. The hex representation of the keypress is only available in the X register if GK is used. IF GETKEY is used, SAVER is called and RESXAF is used afterwards. This means that whatever the contents of the X register was before GETKEY is called will be the same afterwards. Only GK leaves hex in X.

Since the keyboard I/O routine used the onboard beeper, it might be appropriate to discuss the BEEP routine here.

BEEP begins with a JSR to SAVER, then configures Port B on the 6532 as an input for bits 0, 1, 2 and 3. It also sets bits 0, 1, 2 high (onboard pullup resistors cause a high here). This causes selection of U37 output 7 to which the beeper buffer transistor is connected.
Bit 3 is used within the routine to enable and disable the output selector U37 which produces a square wave audio output to the beeper based on the timing constants used within the routine itself.

The X register holds a duration constant (loaded at 897A) which specified how long the beep will be active during a selection. To produce an output, we load A with a value 8 and store it at location A402. This disables the output (forces pin 7 of U37 high). Then a JSR to BE2 takes place. This is a delay loop which takes 132 cycles to accomplish. (2 cycles for LDY, plus 26 loops of DEY (2) combined with BEQ (3 cycles) back to the DEY). After this delay is over, we load A with a 6 and store at A402. This enables the output to U37 forcing pin 7 (output 6) of U37 low. Then the delay routine BE2 is used again. The cycle repeats until a total of 70 hex cycles have occurred.

The beep routine can also be used as a general purpose delay by using JSR NOBEEP instead of JSR BEEP. This configures the output port differently and only makes use of the routine itself to waste time.

The BEEP routine was modified by Synertek Systems to form a higher level output than originally used in SUPERMON 1.0. The beeper itself is not actually an audio speaker. Rather it is a transducer. As such, it has its highest output when operating within a specific frequency range.

You may select to use your own version of the beep routine by including in your program the following type of code. This will demonstrate the available frequency range and you may make your own choice of how to modify the routine.

BEEPER DEMO

```
0200   20  86  8B   DEMO     JSR   ACCESS
0203   A9  0D       NEXT     LDA   #$0D
0205   20  A5  89            JSR   CONFIG
0208   A2  70                LDX   #$70
020A   A9  08       DEMO2    LDA   #08
020C   8D  02  A4            STA   PBDA
020F   20  00  03            JSR   NEWBE2
0212   A9  06                LDA   #06
0214   8D  02  A4            STA   PBDA
0217   20  00  03            JSR   NEWBE2
021A   CA                    DEX
021B   D0  ED                BNE   DEMO2
021D   20  9B  89            JSR   NOBEEP
0220   20  9B  89            JSR   NOBEEP
0223   20  9B  89            JSR   NOBEEP
0226   20  9B  89            JSR   NOBEEP
0229   20  9B  89            JSR   NOBEEP
022C   20  9B  89            JSR   NOBEEP
022F   4C  03  02            JMP   NEXT

0300   A0  1A       NEWBE2   LDY   #$1A
0302   88           N2       DEY
0303   D0 FD                 BNE   ~~NEWBE2~~ N2
0305   60                    RTS
```

After loading this demo, begin at location 200 (GO 200 CR). Try changing location 209. This will vary the duration of the tone. Try changing location 301. This will change the frequency of the tone.

Note that as specified, JSR NOBEEP could be used as a general purpose delay loop and has been used as such here - 6 uses of delay from 021D through 022E.

Regarding such uses of general purpose delays, you'll note that during the delay period, the display remained blank. If it doesn't matter whether the display is active during the delay period, we could instead make use of the DELAY subroutine.

To do this, in place of the code at 021D in the beeper DEMO, use the following:

```
021D    A9  09          LDA     #09
021F    8D  56  A6      STA     TV
0222    20  5A  83      JSR     DELAY
0225    4C  03  02      JMP     NEXT
```

and run the DEMO again. Now during the delay period, the display shows whatever was last placed into the display buffer, in this case ".G 200". The duration of the delay is controlled by the byte at 021E.

DELAY itself begins at location 835A. Let's look now at the way it performs. It begins by loading the index register from the Trace Velocity location (A656) and adjusting the value in scratch locations 8 and 9 (SCR8, SCR9) to a value of (FFFF minus (2 raised to the X power)). The vectored scan display routine is used as a part of the delay loop and it stays in this delay loop until 2 to the X display subroutines have been executed.

As part of this delay loop, the keyboard status is checked. If the INSTAT routine finds any key down, the delay routine will be terminated. Thus, if your routines use the delay routine and you have set the delay constant too large, you can speed up the overall execution by repeated key-presses. This may be especially useful during trace of routines using the autotrace function suggested in the SYM Reference Manual, to speed through portions of the program where no problems occur and get to the problem area faster.

This early escape facility may be demonstrated by the following program sequence:

```
0200    20  86  8B    BEGIN   JSR    ACCESS
0203    A9  0B                LDA    #$0B
0205    8D  56  A6            STA    TV          Set Delay
0208    20  5A  83            JSR    DELAY       Delay
020B    20  72  89            JSR    BEEP        BEEP
020E    20  86  83    DLYA    JSR    INSTAT      wait for key release
0211    B0  FB                BCS    DLYA
0213    EA                    NOP
0214    A9  20                LDA    #$20        ASC11 space
0216    20  47  8A            JSR    OUTCHR      To display
0219    20  47  8A            JSR    OUTCHR      4
021C    20  47  8A            JSR    OUTCHR      Times
021F    20  47  8A            JSR    OUTCHR      Total
0222    A5  00                LDAZ   00          Get Display Val
0224    69  01                ADC    #01         Add 1
0226    85  00                STAZ   00          Put it back
0228    20  FA  82            JSR    OUTBYT      Send it out
022B    4C  03  02            JMP    0203        Go back
```

Note: If we are using the early escape from delay in a repetitive sequence, we must also use subroutine INSTAT and the branch back to the INSTAT call as shown in 20E through 212. Otherwise a double escape will take place because the key has not been released. Rerun the program with "EA" in all locations 20E through 212 and see what happens.

The early escape demo uses two monitor subroutines OUTCHR and OUTBYT. Let's take a closer look at these.

OUTCHR begins by saving the registers with SAVER. The TECHO flag is checked to see if a character is to be output to the terminal. If so a jump to (89C1) OUTDSP occurs.

If the ASCII character in register A is the Bell character, a jump into the BEEP routine occurs with no effect on the display.

If not the Bell, it tests for a comma (ASCII 2C). If a comma is specified, it will load the segment content of the rightmost digit of the display from RDIG of the display buffer DISBUF, and will turn on the most significant bit, representing the decimal point.

If the ASCII in A is neither the bell nor the comma, the x register is loaded as a counter and pointer to search a table known as ASCIM1. After each comparison is tried, the contents of x is reduced by one. When (and if) a match is found, the segments to be turned on are found from the SEGSM1, using the current x register contents as a pointer into that table. This segment code is loaded into register A, then pushed onto the stack.

At this point, the DISBUF area is moved around to make room for the new digit. Using the x register as a loop counter and pointer, Digit 2 segment codes are moved to the Digit 1 position, Digit 3 to Digit 2 and so forth. Finally the segment codes representing the new entry are pulled off the stack and stored in the RDIG (Digit 6) position.

Now when the SCAND routine is called, the segment codes for the six scan locations will be output from DISBUF.

When we use the OUTBYT routine rather than OUTCHR, we are attempting to output two hex digits instead of one. These hex digits are the hex interpretation of the upper (most significant) 4-bit nibble and the hex interpretation of the lower 4-bit nibble.

OUTBYT begins by pushing A register onto the stack twice. The reason for this will be explained later. It then does a logical shift right 4 times. This has the effect of moving the high 4 bit nibble into the right 4-bit nibble position, shifting zeros into the left-most 4 positions.

How about an example. Let's say we're trying to output "5E", using OUTBYT, to the display. The result in the A register up to this point (after the right shifts) is "05". Now we take this results nibble and JSR to NBASOC (8A44) which directs us to JSR NIBASC (8309). The reason for the double JSR will become apparent later.

NIBASC takes the figure in the A register and compares it to "OA". If it is less than OA (00-09) all we need to do is to add hex 30 to the number to convert it to its equivalent in ASCII. Then we can return from this routine. When the compare instruction was used, the carry flag is set if the result of the A-contents minus OA was greater than or equal to zero. This results in a branch to NIBALF (8313) where we add a hex 36 plus carry which results in a conversion from OA-OF to ASCII equivalent in A of 41-46. Then the RTS is executed.

The return takes us to 8A47, which is OUTCHR. Therefore, the first character representing the most significant 4 bits of our byte to be output, has been sent to the display. The RTS at the end of OUTCHR takes us back into the OUTBYT routine at 8303. Here we pull the intended output byte off the stack again, don't do the left shifts, and send it through NBASOC again. This results in sending the ASCII equivalent of the low nibble into the display. In our example, we've now set "OE" to NBASOC and wind up with hex 45 as we enter OUTCHR. As this is the ASCII equivalent for E, we have fulfilled the goal of OUTBYT, sending 5, then E into the display.

Now that we've analyzed the basic I/O routines for keypad/hex display communications, let's look at some of the ways the monitor circuit uses them.

COMMAND INTERPRETATION

Commands are input to the monitor in the GETCOM subroutine. At 80FF, the display line is cleared by the transmission of a carriage return-line feed (JSR CRLF). Then the prompt character, a period, is output to the display.

INCHR subroutine is called (at 8107). This cycles through the input vector at INJINV (called from 8A1E) to receive an ASCII character from either the terminal or the keypad, whichever routine address is specified at the input vector locations A661 and A662.

At 8A21, whether the ASCII data was obtained from the terminal or the display, we still can treat it the same way: Remove the parity bit (if any) This is the most significant bit so AND-7F is done. Then we look to see if we have received a lower-case ASCII character (lower case ASCII a-z are 61-7A hex). Rather than have separate routines to handle upper and lower case letters, we convert the lower case to upper case letters by dropping the "5" bit thru an AND-DF at 8A2B.

Now we find out if the character was a Control-O (control key plus the letter O together). If it was, it causes a change in state of the bit 6 of the TECHO flag. If this flag is a 1, there will be an output to the display during command execution. If this bit is off (a zero), it inhibits output to the display.

Just before we exit the INCHR routine, we compare the incoming character to the carriage return ASCII (0D). Then we use RESXAF to go back. Therefore, the flags will be set by the compare and carried intact back to the calling routine.

If the zero flag is set in the return to 81DA, it means the input character (command-input) was the carriage return. We will therefore branch back to 80FF, output another carriage return, line feed and prompt character Then we will await another command.

If a null character or delimiter character (00 or 7F) are received as input, we ignore them and go to look for an executable command. If we find an L, S, or U, we have to "hash" together this input and the next input character to form the SYM command hash equivalents for L1, L2, S1, S2, or U0-U7. (If these were keypad inputs, the hash codes would have been received directly, but we need two keys to express the same thing if coming from a terminal).

The hashing of the load commands at 812F is accomplished by taking a byte 01, getting and adding the next character from the terminal (in this case an ASCII 1 or 2 (31 or 32) resulting in 32 or 33 hex. Then we strip off the left 4 bits (AND-0F) then (OR-10) insert the last hash bit. This results in either a hash 12 or 13 which are L1 and L2 commands.

The hashing of the S or U commands is a little different. We take the ASCII S or U (53 or 55) and shift left twice (result 54 or 4C). We store the result in LSTCOM, get another character, adding in LSTCOM, strip off the left 4 bits and add on the hash bit. This results in the codes for S1, S2, (1D, 1E) or U0-U7 (14-1B).

Then at STOCOM (store command) we put it at LSTCOM (last command). Now we do a JSR SPACE which sends an ASCII code 20 (blank space) to the display. Now we do a JSR to PSHOVE twice, followed by a jump to PARM. Now that we have received a command, we must go and get zero to three parameters.

The subroutine PSHOVE shifts the input parameters down one position each time it is called. Specifically parameter 3 moves to parameter 2 position while parameter 2 is moved to parameter 1 position and parameter 1 is shifted out and lost entirely. Parameter 3 becomes zero.

These are all 16 bit parameters, each occupying two 8 bit locations. The PSHOVE routine uses the index register to loop through the bit shift instructions from 820A to 821C a total of 16 times. The initial shift is an arithmetic shift left of P3L, the low byte of parameter 3. This shoves the leftmost bit of P3L into the carry bit and a zero into the rightmost bit. The rest of the bit shifting for these parameters is done using ROL (rotate left) instructions. The carry bit enters the rightmost position of the next byte and the leftmost bit enters the carry during the shift. Therefore 16 shifts will entirely move each parameter as noted above, leaving a zero in the last (P3) position. Since we called PSHOVE twice, we'll enter PARM with zeros in the P3 and P2 positions.

At PARM (8220) we use SAVER to save the registers. Then initialize PARNR (parameter number) to zero. Later, based on this number, we will jumpt to specific areas to process commands having 0, 1, 2 or 3 parameters. Now we use PSHOVE again to zero out the P1 position as well.

Now its time to fetch parameters. INCHR is called to get the characters. If the input charcter is a delimiter (parameter separator) such as a comma or minus sign, we see if we have already processed 3 parameters (at 8244). If not, we shove the existing parameters down one position with PSHOVE. (Has the same effect as entering 0000 as a parameter followed by the delimiter).

If a hex character (0-F) is entered, it is converted by ASCNIB to a low nibble (4 bits) and all 4 bits are shifted a bit at a time into P3L while the existing bits in the leftmost 4 are shifted into P3H. This allows us to delete leading zeros on parameter entry since there will be zeros in each parameter to begin with. We are shifting in the data from the right and pushing leading zeros ahead of the data as we shift it into each parameter. The nibbles are continiuously shifted until a delimiter or the carriage return is sensed.

To verify this, you may try the following. Store the following program:

```
0200 20 72 89   GX  JSR   BEEP
0203 4C 00 02       JMP   GX
```

Then try entering:

```
GO 88880200 CR
```

This will beep the beeper continuously because your actual GO command is interpreted as GO 0200 CR. So if you make an error during a parameter entry, just keep hitting hex (0-F) keys until the correct digits appear as the last 4 entered, then use the delimiter or carriage return normally. Also if a mistake has been made, make sure you use the leading zeros for the correction since the parameters are each defined by the last 4 digits input.

As we sense each of the delimiters, we add one to PARNR. When the carriage return is sensed, we close out the GETCOM routine leaving the carriage return character in accumulator A. If by chance we were forced back to the calling routine by the use of a non-hex character sensed in ASCNIB, the ASCII for that character would remain in the accumulator and we would enter DISPAT with something other than OD.

If something other than OD is in the accumulator, it means we have entered something like "D5GO". This does not match the normal command/delimiter sequence defined by the monitor. It therefore jumps to the unrecognized sequence vector which, as a default, directs us to the ERMSG routine. If we had invented a new command sequence, to direct the monitor into this, we would change the address at A66A, A66B to point to our own routine which checks for this command sequence.

The DISPAT subroutine does nothing more than load the accumulator with the last command entered (LSTCOM) and load the index register with the parameter number PARNR (says how many parameters were entered with the command). Then it jumps to to BZPARM (0 parameters) or B1PARM or B2PARM or B3PARM for 1, 2, or 3 parameters entered. Each of these execute blocks are constructed in a similar manner in that each compares the number in the accumulator in turn to each command it understands, then either begins to execute it or branches to another comparison. If none is found, it jumps to the Unrecognized Command Vector (stored at A66D, A66E). The default address is again the ERMSG routine but as with the Unrecognized Syntax Vector, we can store our own address at A66D, A66E, (low byte, high byte) and thereby invent our own commands.

Since we've now established the methods used to access the keypad commands, we'll proceed to examine the routines individually. We'll begin with the zero parameter commands.

ZERO PARAMETER COMMANDS

The first one we encounter is R - the register display command. At 8399 we start with a carriage return-line feed, then send "50" to the display, an ASCII P. A JSR SPACE follows, sending an ASCII blank to the display. Then we use the OUTPC routine to pull the current program counter (user PC) from its temporary storage space (A659, A65A), the high byte, then the low byte, sending each to the display using OUTBYT. On this initial display for the program counter, the last item we send to the display is a COMMA through COMINB.

Next it gets a keystroke from the keypad or terminal (must be hex 0-F keys). Then gets a second keystroke, using ASCNIB each time it assembles both into a single byte in A.

On the return, this byte is stored in scratch location 4 (SCR4) and INBYTE is called to obtain two more hex keystrokes. The combination of all 4 strokes represents a user-desired change to the program counter contents and these entry digits in the SYSRAM. On the exit from the monitor (via a CR) the register contents present at the time this routine is entered will be pulled from the SYSRAM and will again become the contents of those same registers on return to the user program.

To clarify this further, when a BRK/NMI/IRQ interrupt is issued, the contents of all of the registers are placed in temporary storage locations during the appropriate interrupt service routine. At the end of the interrupt routine it takes those registers from the places in the memory where they were stored and puts them back. So if we change those numbers while they're in the memory, the return from the interrupt will put those new numbers in the registers for you.

Having seen how we could use the R command to change the user-PC contents, we have to understand what this means. We are changing the program counter contents to tell the processor to continue the program at a location other than the location it would have gone to on its return from the interrupt. Thus using break commands, we can manually alter the sequence of program operations through the change to the user-PC during register display.

We've gone through the program counter display function. This is the only 4 digit display/4 digit change routine in this sequence. Therefore, it was kept separate. The other registers, being only two hex digits, can be displayed in a loop from 83D5 to 83F2.

In each case, the routine begins by retrieving the name of the register (from RGNAM) and sending it to the display, then sending three blank spaces to the display. Any change bytes are input and placed in their appropriate spot in the SYSRAM just like the user-PC sequence above.

The Y register is used to control which register is being displayed. Indexed addressing modes are used for retrieval of register names and for placement of data within the SYSRAM.

After each register is displayed and/or changed, the Y-register is incremented one which points to the next register name and the next register content within the group. We display stack, flags, A, X, Y, then the current Y reaches 6, forcing us back to the program counter display again.

After each character is input from the keyboard, we use the ADVCK (Advance-Check) subroutine to determine whether the input character is the forward-arrow or an ASCII blank (hex 20) has been input. If it has, the routine advances to the next register display. Receipt of a CR character will cause an exit - returning to the calling routine.

The next routine in the zero parameter block is the GO command. The zero parameter GO is primarily used during DEBUG operations to follow program execution (<u>outside</u> of the monitor circuit) one step at a time. This restores all registers to their former contents before the interrupt and program execution continues at the location pointed to by the program counter.

The paper tape load routine follows next in this zero parameter execute block. This routine is covered in a separate section, along with the save paper tape subroutine.

The next one encountered is the DEPOSIT. With zero parameters, it jumps into the one parameter DEPOSIT routine at NEWLN (new line - 84E1). This will follow the normal DEPOSIT sequence, using the current contents of 00FE and 00FF as the pointer to where the data will be deposited. The MEMORY and VERIFY zero parameter commands also assume the old FE, FF contents are to be used. These are explained in the next section on one parameter commands, but before the jump (to VER1 + 4), VERIFY takes the contents of FE and FF and puts it into P3L and P3H for further use within the one parameter VERIFY command.

The final two zero-parameter commands are the L1 and L2 commands. These are the load KIM or load HISPD tape records. The Y register is set up to indicate tape mode. Then a jump to the tape routine is executed. The zero parameter section concludes at 84D7 with a jump to A66D - the Unrecognized Command Vector. As with the Unrecognized Syntax Vector, the default jump is to the error message routine. However, we can change this vectored address at A66D, A66E to point to our own command interpreter if not recognized by SYM.

ONE PARAMETER COMMANDS

Before discussing the one parameter commands, we must take note of the methods used in entry and access of the parameters. From a standpoint of a basic description, one might expect P1H and P1L to be involved in a one-parameter command sequence in that they are parts (high byte, low byte) of parameter one.

This is not true. The single parameter commands involve P3H, P3L. The two parameter commands involve P2H, P2L as the "first" one, and P3H, P3L as the "second". Of course the three-parameter commands have P1H, P1L as the first, P2H, P2L as the second, and P3H, P3L as the third parameter. This should be kept in mind through the discussion of these 1,2 or 3 position command sequences.

The reason this does not match the "basic description" lies in the manner in which the shifting of parameters takes place during PSHOVE. If you go back to that description, you'll find they are set up as

P1H	P1L	P2H	P2L	P3H	P3L
00000000	00000000	00000000	00000000	00000000	00000000

where each new parameter enters P3H and P3L and PSHOVE moves everything left 16 bits to make room for another parameter to be entered.

Back to the subject at hand, one parameter commands. The first one we encounter is DEPOSIT (84DE). The first activity is a JSR P3SCR. This subroutine moves the parameter 3 high and low bytes to 00FF and 00FE respectively then returns. This allow the processor to use the indirect indexed addressing mode later in the program. (84F0, 84F2).

Next we output a carriage return, line feed, and the contents of locations FF, FE which are the 4 digits which represent the memory location at which we wish to begin to deposit data. This part is accomlished by the JSR CRLFSZ (8316). As may be seen from following that routine, at 8316 we output the CR, LF, then load X with contents of 00FF and load A with contents of 00FE, and jump to OUTXAH. OUTXAH pushes A onto the stack, moves X into A and calls OUTBYT (causes output of contents of 00FF to the display). Then A is pulled off the stack and OUTBYT is called again (contents of 00FE).

When we return to the calling routine, at 84E4, we set up the X register as a counter with contents = 8. Then we output a space to the display and go to the keyboard to get a byte to be deposited into that memory location. As noted, INBYTE is the routine which accepts and assembles two ASCII characters into a single hex byte.

If we fail to use "hex" keys 0-9 or A-F during this operation, we will exit the routine with the carry set and ASCII for that key in the A register. This combination will jump us from 84EE to 8501 through to 8553 to 8554. The return will cause an error to be displayed.

If we use legal keys, the routine tries to write into the specified location (code at 84F0), specifically add content of Y to word found at 00FE, 00FF and use this as the address which will be written into. Then it compares what is in the A register to what is in the location. If not equal, it means we could not write in the selected location. It is either a ROM or other -

non-writeable area. The monitor sends a question mark to the display (JSR OUTQM), then proceeds to the next operation. If equal, no question mark.

The routine called next (at 84F9) is INCCMP which might be interpreted as increment and compare. We take the byte stored at 00FE and increment it. If the result is zero, we also increment the byte stored at 00FF. As an example, if we were to begin depositing bytes at 02FF, we would want the next byte to be at 0300. So since 00FE contains the low byte of this address parameter and 00FF the high byte, you can see that this method will correctly increment our address.

We have to decrement the X register (at 84FC) and determine if 8 bytes have been input. If so, a new line is needed since we have output an address to the display and 8 bytes with spaces between them. (see examples in the Sym Reference Manual).

One last option this routine allows is a test for an ASCII space (hex 20) as an input. This does not cause any character to be stored and simply skips us to the next memory location.

The routine is ended by receipt of a CR character (ASCII 0D) which causes a fallout at locations 850E, 850F, following the compare instruction at 8503.

The MEMORY routine (8514) starts in the same way as the deposit, by output of CR, LF, then the start address from parameter 3 (the one parameter entered), setting up 00FE and 00FF as well. Then it outputs a comma at 851A (on the SYM hex display, a comma is a decimal point, which, in this case, separates the memory location information from the data byte).

Now we load A with the contents of the byte located at the address contained in 00FE (lo) and 00FF (hi). A JSR To 0BCMIN (at 8521) takes us to 81D3 where, as the comments state, we output a byte (from A), output a comma (turn on Decimal Point of RDIG) then input a byte. Again a legal byte combination of hex characters must be input or else a return with an error will occur.

Then we store that value just input and verify it. (At 8528, 852A). As with the deposit routine, we output a question mark if the verify failed and continue on with the next location.

INCCMP is used here also to allow the advance to the next location. However, in this case we have a number of ways to get there. Specifically, a forward arrow (hex 3E) or a space bar (hex 20) will advance it to the next location.

If we used a reverse arrow (hex 3C) the machine would go to PRVLOC (previous location - 8555) and call DECCMP instead of INCCMP. This DECCMP, at 82BE does a double byte subtract-one on the contents of 00FE and 00FF, just as INCCMP was a double byte add-one. This moves our pointer to the previous memory location, and we jump back to NEWLOC through the display output routines again.

The use of a plus-sign sends us to LOCP8 (855B) which is a double byte add-8 to the contents of 00FE, 00FF. A minus sign in turn sends us to LOCM8 (8569) which is a double byte subtract-8 from the contents of 00FE, 00FF. Then we go to NEWLOC (8517) where the pointer now shows either a location 8 bytes more or 8 bytes less than the previously displayed address.

As with the DEPOSIT, a carriage return will cause an exit to the calling routine (Return from JSR OBCMIN at 8524 will cause a jump to NH42-8537, then to EXITM1-8577 and return if hex "0D" had been the input character - the CR key).

The next one-parameter command is "GO", routine beginning at 857D. A CR, LF is sent to start the display on a new line. Then it uses JSR NACCES to set the write protect bit on the system RAM area.

Since the 6502 has a limited stack area, this routine tries to allow the maximum possible space to the user. So it loads X with hex FF and transfers X to the stack pointer. The stack value now is 01FF. Next it pushes the (return address -1) onto the stack. In this case it is 7FFF in hex.

Then it pushes parameter 3 hi and P3L bytes onto the stack, loads A with the flag register contents stored in location FR (A65C) and pushes this onto the stack. Then restores X, Y and A registers to their original contents from locations XR, YR, AR and executes a RTI (return from interrupt).

The RTI has the effect of pulling the flag register contents from the stack and restoring them to the flag register. Then pulling the next two bytes off the stack making that the contents of the program counter where the next instruction is to be found. For the "GO" routine, this points to the first location of the user program.

Now would be a good time to emphasize the difference between RTS and RTI. You noted above that in an RTI execution, we pull the flag register contents off the stack, then the new program counter contents.

For an RTS, the flags are not involved. Instead two bytes are pulled off the stack and put in the program counter. Then the program counter is incremented by one and the resultant value is the address where the next instruction is located.

Using the GO routine itself as an example, you'll recal we pushed "7FFF" onto the stack before heading into the user's program.

If, at the very end of a user's program, an RTS is executed, we would pull 7FFF off the stack (assuming an equal number of pushes-pulls JSR-RTS before this occurred). The RTS action increments this address giving us 7FFF +1 = 8000 where it finds its next instruction. This is the cold start entry into the monitor.

Now we come to the one-parameter VERIFY (859A) routine. This is to output a checksum for a total of 8 bytes only beginning at the address specified in parameter 3H, 3L. Rather than write a separate routine for this, including output controls and checksum routines, this one parameter routine moves P3H into P2H and P3L into P2L, then adds hex 7 to the original value in P3H, P3L and stores it in P3H, P3L. The result then looks like an entry of two parameters for the verify routine, which then allows a jump <u>into</u> the two parameter verify at 8640. See that description for the way the verify ends.

Following VERIFY is the area which handles the JUMP command (at 85B8). Only JUMP 0 through JUMP 7 are valid commands. The addresses of the areas called by the JUMP are contained in the system RAM at A620-A62F.

Since these are all two byte addresses, to do a table search and jump, any comparison must be by increments of two so that we point to the correct location. Let's look at how the routine accomplishes this correctly.

This value is put in the Y register and is used as a pointer into the JUMP table (JTABLE). Since each value possible in Y is a multiple of two bytes, we will point to either the hi byte (JTABLE+1, Y) or the low byte (JTABLE, Y) of the selected address no matter which value 0-7 had been in the accumulator in the first place.

To do the jump itself, we follow the same steps we used for the GO routine above. Specifically restore the stack value to 01FF at 85C4, push 7FFF on the stack (translates into 8000-monitor cold start on RTS), then load into A the hi byte of the jump address, push onto stack, load lo byte, push on stack, then finally go the NR10 (8408) which is the same sequence followed at the end of the one parameter GO routine as described earlier.

The next single-parameter routine (at 85D7) is the tape-load. If KIM format (L1-hex 12) we must set KIM mode LDY #00, then be sure that ID byte selected is not FF (KIM format tape load does not provide for program relocation - in a one parameter only sequence, must reload into same location from where it was dumped). Then move the parameters around, P3 into P1 position, and jump to the tape load routine.

If we're in the SYM format load (one parameter), we will load the tape mode into Y (LDY #80), move P3 to P2, then jump to the tape load routine. A separate description of the tape routines occurs later in this book.

The second last of the one-parameter monitor routines is the Write-Protect RAM. We can select to write-protect 1, 2, or 3. These numbers represent the area from 0400-07FF (1), 0800-0BFF (2) or 0C00-0FFF (3).

This routine takes the parameters P3H and P3L and moves the bits around until they are within a 3 bit range which controls the actual write protect. Specifically, the entry might be W 101 CR, meaning Protect 400-7FF, unprotect 800-BFF and protect C00-FFF.

This routine takes P3H and P3L, represented in <u>hex</u> by 0101 which is 0000 0001 0000 0001 in binary. These bits then we send to output register 3A (location AC01). This will cause the appropriate segments to be protected <u>if</u> the jumpers are installed at JJ-42, KK-43 and LL-44.

The last of the one-parameter commands is the CALCULATE function. Since this is a one, two or three parameter command, we'll save it for later in the three parameter area.

TWO-PARAMETER COMMANDS

The first one we encounter is STORE-DOUBLE-BYTE. This is used for changing address vectors within memory. We think of addresses as we might normally encounter them - high byte then low byte. However, the processor, as it increments the program counter, must find the low byte first, then the high byte to form the complete address. This routine allows us to specify the address vector location and its location in the "ordinary" logical 4 digit sequence and has the processor take and store it away in the sequence (lo-high) which it will use itself.

At 861D it calls P3SCR subroutine to initialize 00FE, 00FF to the value in parameter 3, then stores parameter 2H, 2L in the selected locations using the indirect indexed mode in the correct order. A return to the calling routine occurs on completion.

The next two-parameter command is encountered at 8633. This is the memory search command. The function of this two-parameter version of the memory search is to begin searching the memory for a specified data byte. The search starts at the location pointed to by the bytes stored at locations 00FE, 00FF and ends with the location shown in the second parameter entered.

As we locate this byte, we can step forward or back from the point to examine or modify this, or surrounding memory locations.

To clarify this further, a specific example is in order. Try the following sequence:

```
M  200  CR  11  22  33  44  55  66 77  88 minus-sign  RST ....then..
M  44-220  CR.
```

What we did was to fill locations 200-207 with bytes 11-88, then step back 8 bytes to location 200 to see what we've done. When we are in the memory display routine, (refer back to one parameter MEM command), we are storing the address of the data in location 00FE (low byte) and 00FF (high byte). Then we use this data from these locations to point to the data byte to display via the indirect indexed mode.

When we hit the reset (RST) key, the data in 00FE and 00FF is not disturbed. Now if we come back to the two parameter memory command, we will treat it like a three parameter memory command instead. Specifically, we will begin a search for byte "44" ending at location 220 and beginning at the location already specified (in this case 200), where the beginning address is stored in locations 00FE, 00FF.

Throughout the monitor program, these same locations and this same addressing mode is used for data retrieval and display. Therefore, these same two parameter memory search and display techniques may be used in combination with many commands.

Just to confirm this, with the data still in place at 200-207 from the previous example, try the following:

```
SHIFT DEPOSIT  200  CR  00  CR
        MEM    44-220  CR
```

This treats the MEM command as a search for byte 44 beginning at 201 and ending at 220. You will note that the effect is exactly the same as in the first example.

Also try: GO 200 CR. This forces a program controlled break displaying "0201.0". From this point, the sequence MEM 44-220 CR again causes the same display and operation as in the preceding two examples

In the two-phase evaluation routines for the memory function, at 8633 the low byte of parameter 2 is copied into the low byte of parameter 1. Then a jump into the 3-parameter memory routine takes place. As you will notice, the point of entry at 8808 is below the JSR P2SCR which would have moved parameter 2 into 00FE, 00FF. So we have left the original value in those locations, allowing for the operation described above.

Now we come to the VERIFY command. As noted earlier, the zero parameter VERIFY command was converted to a one-parameter verify by picking up the starting address out of 00FE, 00FF. Then in the one parameter VERIFY area, we saw the one parameter entry converted to a two-parameter entry, evaluated here by the two-parameter command sequence.

As a reminder the function of the VERIFY is to add up all of the hex values of data bytes between the addresses entered at the first parameter through and including the byte at the second address entered.

The first action taken (8640) is to move parameter 2 into 00FE, 00FF. (Another reminder - two parameter commands use P2 and P3, not P1 and P2). Then we do a JSR ZERCK which is to zero the checksum scratch area (SCR6, SCR7).

Now a loop is set up (8649) using the X-register as a counter. At 864B output a space. Then load the byte pointed to by the address stored at 00FE, 00FF into the A-register. The CHKSAD (82DD) subroutine is called to add the current contents of the A-register into the checksum.

You will note that CHKSAD uses the system stack. This leaves the byte in the A-register on its return. We then call OUTBYT to display this byte after adding it to the checksum.

Next, INCCMP (increment word at 00FE, 00FF and compare to parameter 3) is called. If we have added up all bytes between the two limits specified, a branch to V1 (866E) is made. If not, we will continue to add bytes to the checksum and output them until 8 bytes are output (controlled by the X-register loop). Then a comma is output, followed by output of the low byte of the checksum. The comma, then checksum low output is accomplished by the JSR OCMCK.

The routine continues by starting a new line (by the loop back from 866A to 8646 and output of address, space, ((byte, space) times 7), byte, comma, checksum low until the upper limit has been reached.

At V1 (866E) we go in a loop from 866E to 8676 just "using up" the X-register counts with no further output, then output the low byte of the checksum. On the next line, the output of the low and high byte of the checksum will occur. (Called from 8684).

As a special note, if we are using a printer to output the VERIFY routine, each of the lines of output will, as noted above, have at its end the low byte of the total-to-this-point checksum. It will therefore not match the line-by-line checksum which one might generate if these lines were verfied one at a time.

The next routine using two parameters is the KIM format load. At 868C the low byte of P2 is loaded and checked that it is "FF". This ID byte indicates that the tape record is to be loaded into an area different than the area it occupied at the time of the dump. The P3 parameter then will indicate the starting address for the load.

Within this section of the routine, we set the tape mode to indicate KIM format by loading the Y-register with zero, then jumping into the tape routine at location L11D (85E9). (Note that the SYM Reference Manual contains a typographical error in the comment field for location 8493; should be "MODE = KIM"). We'll continue the tape load and save routine description later.

The SAVP (Save Paper Tape) is the last of the specific two-parameter commands. We'll save this command for the paper-tape section which follows later in this manual.

The CALC or EXEC commands can have two parameters also, however, we'll cover those cases in their separate areas under the three parameter commands.

THREE PARAMETER COMMANDS

The FILL command is the first one encountered at 8718. Locations 00FE and 00FF are initialized to the value give in Parameter 2 (JSR P2SCR).

The byte at P1L is the byte we wish to store at all locations between Parameter 2 and Parameter 3 address. Therefore, P1L is loaded into the A register (at 8729) and is stored, using indirect indexed mode, in the address pointed to by bytes in 00FE, 00FF. Then the store is verified by a comparison to the A-register contents using the same addressing mode.

If an error occurs (write into a ROM location or a bad memory bit), the ERCNT (error count) is incremented and the routine continues. INCCMP is used here again to increment 00FE, 00FF to point to the next location and to compare that end value to Parameter 3.

When we finish the FILL, we branch from 8731 to 87AF where the error count is tested. If greater than zero, the error count value is left in the A-register and the carry is set. If the routine has been called by the normal monitor call sequence (beginning at 8006), the ERMSG routine will output this count if the carry flag is set.

One final note about the error count loop is in order. This is the provision for eliminating wraparound. Specifically the FILL routine calls the BRTT (87C1) subroutine. If this was not done, errors in multiples of 256 would never be reported since a change from FF to 00 would occur each time. With this provision, 255 errors or more will be reported as "ER FF".

The BLOCK-MOVE routine comes next, beginning at 8740. It begins by initializing the error count to zero. Then uses JSR P2SCR to move parameter two into 00FE, 00FF. At 8484, parameter 1 is moved into 00FC and 00FD.

Next the most significant address bytes of both parameters are compared to determine which direction the move is to take place. (Byte in A is P1H, Byte in 00FF is P2H). If the carry is set, the result A - (M) was greater than zero. This means we take the branch from 8754 to 875C to 8772. This represents a block move in a positive direction. An example of this would be SHIFT BMOV 0303-0300-0350 CR meaning take the contents of locations 300-350 and move them into 303-353.

If the direction was not calculated, one might have erroneously begun the routine by moving 300 to 303, then 301 to 304, then 302 to 305 and so on. But when we get to moving 303 to 306, we have a problem in that we've wiped out the section of code which we were trying to move. As a result, we would only have duplicated the sequence from 300-302 throughout the whole move-space.

For this example, we had to begin the move at the top of the space instead of the bottom. We would therefore move 350 to 353, then 34F to 352, 34E to 351 and so forth. This maintains the block of data exactly as expected. The code from 8772 through 879C sets up for this type of move.

This specific operation although done byte by byte in this area may be summarized as follows:

Start with	Parameter 1
ADD	Parameter 3
Subtract	Parameter 2
Store results in	00FC, 00FD

Set up 00FE, 00FF from parameter 3

Move parameter 2 into parameter 3 position

Then use the loop in BLP1 (879D) to control the moves. Lets look at the overlap example again so that we have some specific numbers to use.

```
The example was BMOV 303-300-350  CR
Parameter 1 is      0303
Add P3              0350
                    0653
Subtract P2         0300
Result (00FC, 00FD) 0353
         Lo    Hi
Now move 0350 to 00FE, 00FF
                   Lo    Hi
and move P2 to P3 position
```

What we have actually done here was to calculate the 4th parameter of the move input. Once this is done, at BLP1 we call BMOV subroutine.

BMOV loads A with data from the address pointed to by 00FE, 00FF and stores it into the address pointed to by 00FC, 00FD. A verify comparison is made and the error count is incremented before the return if the result of the compare is not correct. On this first pass, it moves data from 0350 to 0353.

The BLOCK-MOVE in the negative direction is more straight-forward in that no special calculating is needed. This loop is contained within the area 875E through 8770. It does a move, returns, increments 00FC, 00FD as a 16-bit word, then calls INCCMP to increment 00FE, 00FF as a 16-bit word and compares it to the contents of parameter 3. It will loop back until the move is complete.

A specific example (with overlap) would be the following - to close up a space in the code if desired:

```
BMOV 300-303-353 CR
```

Because of the initial setup specified earlier, the first move will be from 0303 to 0300. Locations 00FE, 00FF will point to the value held in parameter 2 at the beginning. Locations 00FC, 00FD will have been initialized to the value in Parameter 1.

The loop will increment 00FC, 00FD as a 16 bit word, then call INCCMP to increment 00FE, 00FF as a 16 bit word and compare to P3 value, moving the bytes with BMOVE until the move is completed.

At this point, we encounter the setup for KIM and SYM format tape saving. At 87D1 is the code executed if the KIM format (1D) command is input. This sets the Y-register to zero (for KIM), and assures that the ID byte is not 00 or FF, then increments parameter 3 and jumps into the "SENTRY" point in the tape load routine. (This is the SAVE-ENTRY point).

If the command had been 1E instead of 1D, the hispd (SYM) format would have been selected by loading hex 80 into Y. Then it branches back to S13C 87D3) to check that the ID byte is not 00 or FF, then jumps to "SENTRY" as before.

At 87F2, we encounter the setup code for the 3-parameter SYM format LOAD command. In this case we are trying to load a tape record into an area different from that which it originally occupied.

The ID byte requested must be FF, even though the ID byte on the tape is not FF. If this is the correct first parameter, we branch to 87F9 which increments parameter 3 (JSR INCP3). Then load Y with hex 80 to define high speed mode and jump to "LENTRY" (8C78) which is the LOAD-ENTRY point of the tape routines.

The tape load and save routines themselves are covered in a separate section, so we'll continue with the rest of the 3 parameter commands here.

We had earlier seen that the two parameter MEMORY command converted itself to a three parameter command, then jumped to the 3-parameter area for interpretation. The point of entry from the two-parameter setup is at MEM3C. The only step not executed in this case is JSR P2SCR which sets up the correct value in 00FE, 00FF from parameter 2. Since the two parameter MEMORY command is to use the "old" value, we need no setup and therefore skip that step.

Beginning at 8808, we load the byte we want to find into A. Then compare it to the byte at the address pointed to in 00FE, 00FF (lo byte of effective address, hi byte of effective address). If it is not equal, INCCMP is called to see if we have searched beyond the upper search limit specified in parameter 3. If not, the 16 bit word at 00FE, 00FF is incremented and the comparison is tried again.

When the selected byte is found, at 881C we call NEWLOC (8517) to display the address and contents of that address as already explained in the one parameter MEMORY command. Now, while executing NEWLOC, all normal actions of the one parameter MEMORY are valid, but in addition we can use the GO key. This key results in a continuation of the search for the specified byte beginning at the current location displayed with the ending point still the same. (Once we get to the end pont, we return to the calling routine). Refer to the one parameter MEMORY routine description for further details.

The CALCULATE routine comes next. This begins at 882B. It is very straight-forward and is simply a 16-bit addition and subtraction leaving the result of P1 + P2 - P3 in the X-register (8 bits-hi) and A-register (8 bits-lo). At the end of the routine, output of the result is by a JSR OUTXAH. This routine calls OUTBYT for the X-register contents, then OUTBYT again to output the A contents to the display.

The last of the 3 parameter commands is the EXECUTE command. IF two or three parameters are entered, the second and third are ignored by the routine itself, but are stored in the Parameter 2 and Parameter 3 area where our own routines could use them later.

The standard version of the EXECUTE command uses one parameter. This parameter represents the address where a command sequence is stored. The sequence itself is stored as a series of ASCII characters which will be called in by the monitor one at a time and executed as though they were keyboard inputs.

Before we examine the routine, lets just take a quick example. Store the following data in locations 200 and up.

```
0200  46  41  41  2D  33  30  30  2D
0208  33  46  46  0D  56  33  30  30
0210  2D  33  46  46  0D  42  32  38
0218  30  2D  33  30  30  2D  33  37
0220  46  0D  00
```

```
This translates into:  Fill    AA - 300 - 3FF (CR)
                       VERIFY  300 - 3FF (CR)
                       BMOV    280 - 300 - 37F (CR)
                       Return to normal entry
                       from keypad
```

Now enter EXEC 200 CR and the above program sequence will occur just as though it was entered directly from the keyboard. Any keystroke sequence can be called in this manner, but you cannot EXEC an EXEC command within this block (no nested EXECS allowed).

When you try "nest" EXEC commands, the monitor will not execute them in the expected manner. Just to illustrate exactly what occurs, with the above data in place, also store away the following:

```
0250  45  32  30  30  0D  45  32  30
0258  30  0D  45  32  30  30  0D  00
```

Translated from the ASC11, this would be:

```
EXEC  200  CR
EXEC  200  CR
EXEC  200  CR
Return to monitor
```

So try (from the monitor) EXEC 250 CR. Note that this takes the same amount of time to execute as EXEC 200 CR above. But we should have executed it <u>three</u> times, right? And this should take 3 times as long! This illustrates that execs cannot be nested.

Let's examine the EXEC routine itself to see why this is true. We'll begin with the case where no vectors have been modified and all inputs are coming from the hex keypad.

Starting at 8855, the monitor looks at the input vector high byte at A662 and compares it to the high byte of the execute vector at A673. If they are equal, it means we were trying to nest two execs. This has the effect of an unconditional branch as we'll see in a moment.

Beginning at 885D and continuing through 8865, we will store away the input vector in monitor scratch locations. Why? We are using a hex keypad or a CRT or TTY terminal. When we logged into the SYM, the input vectors and output vectors were configured to point to specific monitor routines for each type of device.

The next step at 8866 to 8871 is to change the input vector to point to the RIN routine (887E) in place of whichever input routine was in efect when EXEC was entered. Then parameter 3 is moved to 00FA and 00FB to be used as a pointer later. This will tell us where the next input byte will be stored, to be used as though it was a keyboard input.

Looking at RIN, we see the indirect indexed mode used again to pick up the byte pointed to by the address stored in 00FA, 00FB, If the byte is non-zero, 00FA and 00FB are incremented as a 16-bit word, the "input character" is echoed to the terminal if TECHO bit is set, and it returns to the calling routine with the character in the A-register.

If the byte was a zero, the input vector is restored from the scratch locations and INCHR is called so that we can grab the next character from the input device rather than the RAM.

This gets us back to the reason for an inability to next EXEC commands. Specifically we only have one set of scratch locations to use to store the "old" input vector when calling for RAM input. These locations must be protected from over-writing after the first EXEC command so that we can restore them correctly after sensing a zero byte. (Otherwise after a restore, we'd still be pointing to the RAM as the input device).

To protect our return address, then, we branch from 885B to 8872 if the input vector had already been moved (and we are therefore trying to execute an EXEC within an EXEC). At this point, the new P3 within the second exec becomes the new 00FA, 00FB pointer. This has the effect of a direct branch as illustrated by the example we tried earlier.

PAPER TAPE I/O ROUTINES

The LOAD PAPER TAPE (LP) and SAVE PAPER TAPE (SP) commands are grouped together here because of the fixed format that the paper tape uses. If we know how the paper tape is punched in the first place, the reloading technique for the tape will be easy to follow as well.

The SP command accepts two parameters. The first is the starting RAM location for a paper tape dump (stored as Parameter 2). The second is the ending RAM location (stored as Parameter 3).

The routine begins at 869D, saving the registers. Then it sets up 00FE, 00FF from Parameter 2. At 86A3 is a JSR DIFF2.

At DIFF2, ZERCK is called to zero the checksum (16 bits). Then the routine follow through the DIFFL before the return. This calculate the difference between the current value in 00FE, 00FF and the ending address contained in Parameter 3.

If the result is greater than zero, we branch from 86A6 to 86AB where we test to see if the maximum number of bytes per pape tape record have already been transmitted. If so, we will go to SP2F (86BA) and output the following to the tape punch: A colon, followed by the number of bytes contained in that record, followed by the starting address (two bytes) of that block of data, followed by the data itself.

The colon in this case serves as a record delimiter, pointing to the start of the record. As such, it is not added into the checksum. The subroutine SVBYTE is used to output all other data to the paper tape, adding each byte to the checksum along the way.

Each of the data bytes is output by the loop extending from 86D2 through 86E7. In each case the loop is controlled by the initial value set to variable RC (count of bytes in that record). This will always be the value 10 hex (MAXRC default at A658) until a number less than this is left as a difference between P3 and the current value in 00FE, 00FF. The last record will then hold the remainder of the bytes in the dump.

When the last byte of each tape record has been output, the checksum bytes, hi then lo will be output using OUTXAH. When the dump is ended, we are required to add ;00 CR to the tape in the offline mode. This is <u>not</u> part of the SP routine. You will notice that the output routines used are OUTCHR, OUTBYT and OUTXAH. To punch a paper tape our output vector will most likely be pointing at the TTY routine. The TTY itself then, not only received and prints the characters, but also punches the paper tape if the punch is on.

The LOAD PAPER TAPE routine follows a sequence to read and store the data in the same order as it was stored in the first place.

We start at 841E saving the registers and output a carriage return, line feed. Then we zero the error count and the checksum. Then it will begin to read tape until it first encounters a semi-colon (loop from 842C to 8432).

At 8433, the first byte after the semicolon is read. If non-zero, it branches to NUREC (new record - 8443) where it stores this first byte at RC (bytes in this record). Then the next two bytes are loaded to set up 00FE and 00FF as pointers to where the data will be stored.

The loop beginning at MORED (more data - 8454) uses the count stored in RC as a control for how many bytes will be loaded from this record.

During the store operations, we also check whether the store operation was good (845D) and increment the error count if any errors occur. While the load is in process the subroutine LDBYTE is used to read the next byte with INBYTE and add its value to the checksum using JSR CHKSAD.

After decrementing RC to zero, it indicates that all bytes of that record have been loaded, then the checksum is read from the tape and is compared to the checksum calculated during the load. (Each record has two checksum bytes). The monitor loops back here to LPZ (8429) to zero the checksum and begins to look for the start of the next record (next semi-colon).

At the end of the load, the end-of-the-file record which was manually added to the tape, is sensed. Here's how that part works:

The semicolon is sensed at 842F, causing the next byte to be read. This byte is a zero (00) indicating that there are no bytes in this record. This causes fall-through to 843A where the error count is checked. If no errors occurred we end up at 8440 which is JMP RESXAF. This is our exit to restore the registers except A and Flags.

When the return to the monitor occurs, if the paper tape reader is still active, the next INBYTE sensed will be a CR (carriage return) character sending the monitor into the warm-start loop. This was the last character of the end-of-file record and ends the routine.

Now that we know that each tape record has a semi-colon, byte count, starting address, data content and checksum, can you guess why the end-of-file record (;00 CR) is added manually instead of being added by the monitor after each dump? Right! The paper tape can dump non-continuous blocks of records, then load them back in with a single command. Example:

```
SP  200 - 215  CR
SP  310 - 350  CR
SP  100 - 140  CR
SP    0 -  25  CR
plus manual entry ;00 CR
where reload is accomplished by:
LP  CR
```

TERMINAL I/O ROUTINES

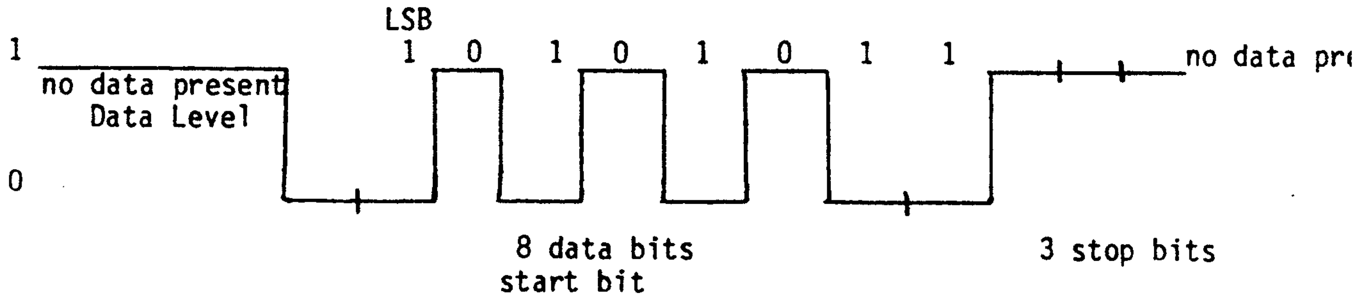
The paper tape routines were based on the ability to talk to a paper-tape reader-punch device, most likely attached to a teletype.

Before going into the routines themselves, we must first understand the basic different between the Hex Keypad and a teletype or CRT terminal. The onboard Hex keypad is a parallel input device. The closure of each key represents an intersection between a row and a column of keys interpreted by the monitor scanning the rows and columns to detemine which key is down.

The teletype or CRT terminal is, most often, a serial type of device, requiring only one input bit and one output bit for full communications. The timing of pulses on these inputs and output bits determines the data content. The maximum rate of change in transitions per second which may be sent by these single bits is known as the baud rate.

For the SYM, the format of the serial input string must be one start bit, 8 data bits, and three stop bits. This is illustrated in the diagram below:

Serial String Data Format



When there are no transitions present, the input line is maintained as a high. The first high to low transition seen is known as a start bit. This stays low for one bit time which is determined by the baud rate. Then, one bit at a time beginning with the least signficant bit, the data is transitted. Last, a series of 3 stop bits are transmitted. This means the data line level returns to a high state for at least 3 bit times before the next start bit comes along.

One thing you will notice about the above diagram is the fact that a continuous "1" level represents a no-signal-present condition. Each place where an actual data bit (1) is to be present, a zero level is the output.

Therefore, if we are to output binary data, it must be inverted before it is output. Likewise after receiving a data bye, it must be inverted before interpreting it. This is accomplished in the input and output routines by an EOR #$FF.

Since we've outlined the requirements for the data format, lèt's see how the SYM will produce this output. We'll assume our output vectors already point to the terminal output routine at 8AA0.

At TOUT (8AA0) the ASC11 character was in the A-register when OUTCHR was called, is saved in location 00F9. The registers are saved and Port B bits 4 and 5 of the 6532 are set as outputs. (These are the CRT and TTY output bit).

Then the ASCII is retrieved from 00F9, the X-register is loaded with hex 0B, and at 8AB1 the data in A is inverted.

From this point onward, the carry bit will be our primary output vehicle, controlling whether there will be a one or a zero sent to the output device. The reason for this statement may be seen by an examination of the OUT routine.

At 8AD4, OUT begins by saving the current contents of A on the stack (it may represent a whole or partially shifted ASCII character). Next it brings the current Port B data register (6532) contents into A and sets the 4 most significant bits to zero using an AND #$0F.

A branch to OUTONE (8ADE) is made if the carry is clear. Note that the AND instruction has no effect on the carry flag. So if it was set on entry to OUT, it will still be set here.

Since the carry is set at 8AB3, we do not branch. Instead we OR the Port B data with hex 30 which sets both the CRT and TTY output bits high when the store operation at 8AE1 is done. If each bit is high if the carry was set, how then does the start bit become a zero? Well, the TTY output is inverted by transistor Q30, providing the expected polarity for the signal. the CRT output at Q31 is inverted, expecting the opposite polarity for all signals. From here onward, we'll just talk about the TTY signals, having standard polarity.

As noted earlier, when we set the carry and did a JSR OUT, it resulted in a low output. This was at 8AB3, 8AB4 in our TOUT routine. At 8AB7 we call DLYF to delay a full bit time beore a new bit is to be output. One more short delay loop from 8ABA through 8ABF is executed, then we get to output the next bit.

Each data bit is shifted out into the carry by the LSR A instruction at 8AC0, the X-register is decremented and a loop back to 8AB4 occurs until X reaches zero. This outputs bit 0, 1, 2, 3, 4, 5, 6, 7, then 3 "zeros" (stop bits, inverted output level = 1) because the LSR A instructions shifts a zero into the MSB each time it shifts the LSB into the carry bit. Then it delays a full bit time for each bit output.

Where did the delay time come from? The particular value loaded in SDBYTE (A651) establishes the delay period. And this SDBYTE value was established at sign-on time. The monitor has to come as close as possible to the bit times of the terminal we are using in order to have the two understand each other. We'll get to that setup later. Before that we'll look at how data is input in this serial format.

The loop at LOOK from 8A5F to 8A69 is provided to watch the terminal input lines continuously until a start bit is located. If there is no start bit forthcoming, at 8A5F the inputs will be zero. This is AND'ed with TOUTFL so that only the 4MSB's will be examined (if TTY is active, TOUTFL = F0).

At TIN (8A6A) we've found the leading edge of the start bit, then we delay half a bit time to put the evaluation roughly in the middle of the bit cell time (in case there's some difference between the transmit and receive "BAUD" rates we allow for some variation by keeping it in the center).

At 8A73, the carry is not set and a subtract with carry will occur. The result of the subtraction will set up the carry bit for an output to the terminal if the TECHO bit is set. It also sets up the carry bit for the sequence beginning at SAVE (8A87). If there is no output to the terminal required (echo) we use DMY1 to delay the routine just as much as the echo would take.

At save (8A87) the ROR A instruction causes the carry bit to enter the MSB of the A register, all other bits move right one position, and the LSB of A enters the carry.

We continue to loop back through TIN until all bits have been received and the stop bits are recognized. Then at 8A9B the received data is inverted and left in A on the return to the calling routine.

This input routine from the terminal operates in the same way as the hex keypad I/O routine in that it stays within the routine continuously (endless loop) until a keypress is detected. This type of action may not be desirable.

The following discussion digresses a bit, showing a set of useful routines for the hex keypad or a parallel ASCII keyboard. Unfortunately, a serial keyboard would not be applicable in this type of activity because of the possibility of mis-timing on the receipt of start bits. Because of this, serial keyboard "fast scan" might have to be done on an interrupt basis instead.

KEYBOARD FAST-SCAN ROUTINES

Using the keyboard during a program without using the GETKEY routine may be necessary where the speed of the program is critical. In particular, we may just want to look at the keyboard to see if a key is pressed, interpret it and go on or simply decide to continue with program execution having found no key down. The following routine illustrates this technique. It is written as a fully relocatable subroutine so that it may be used immediately in any program.

The New CHKKEY Routine

```
0200  20 86 8B   CHKKEY  JSR ACCESS    ; Unprotect SYSRAM
0203  20 88 81           JSR SAVER     ; Save Registers
0206  20 23 89           JSR KEYQ      ; Any key down?
0209  F0 15              BEQ CONTIN    ; Continue previous ops
020B  20 2C 89           JSR LRNKEY    ; Which key was it?
020E  48                 PHA           ; Store it
020F  20 72 89           JSR BEEP      ; Tell user found one
0212  20 23 89   TST1    JSR KEYQ      ; Still down?
0215  D0 FB              BNE TST1      ; Wait for release
0217  20 9B 89           JSR NOBEEP    ; Debounce delay
021A  20 23 89           JSR KEYQ      ; Check once more
021D  D0 F3              BNE TST1      ; Wait again for release
021F  68                 PLA           ; Get ASCII value
0220  4C B8 81   CONTIN  JMP RESXAF    ; Keep ASCII in A, return
```

A Test Program

```
0300  20 00 02  TESTR  JSR CHKKEY    ; Look at Keypad
0303  F0 03            BEQ SCN       ; If none, skip output
0305  20 47 8A         JSR OUTCHR    ; Send ASCII to DISBUF
0308  20 06 89  SCN    JSR SCAND     ; Scan Display
030B  4C 00 03         JMP TESTR     ; Begin again
```

To try the routine, enter GO 300 CR. You will notice that any key-press will have exactly the same effect as it did while the monitor was in direct control. This includes a blanking of display while the key is held down.

The blanking is required during the keypress for two reasons. The most important is to avoid multiple entries where keybounce have have occurred. (The processor executes instructions so fast it may interpret each "bounce" as a separate entry). The other reason, almost as important, is that any keypress during on onboard display scan, messes up the display. This is due to the multiplexed nature of the display as well as the sharing of the display ports with the keypad.

In the February, 1980 issues of MICRO magazine, I presented a routine for interfacing an ASCII keyboard to the SYM. There were some errors which crept in, one in the routine itself and the other in the initializer. Particularly JSR OUTCHR was incorrectly typed as "A5 F1" where it should have been 20 47 8A. The other section improperly translated is INIT in the 6th line, LDA #$40 should have been LDA #$39, pointing to the KYSTAT routine.

The primary problem with the program-as-published was the accidental deletion of the location-column. GKEY, although fully relocatable, started for demonstration purposes at 0200, ended at 023D. The INIT routine, for demo purposes, began at 0240. The vectors in INIT are to point to GKEY and KYSTAT respectively, wherever they are relocated.

If you try that routine as corrected, you'll notice that the display remains lighted even while the key is held down.

The original routine was written by me in August, 1979 and I have since then improved it to operate in the same way as the onboard keypad for more consistent operation. This improved version is presented here.

<u>A Modified ASC1l Keyboard Interface Routine</u>

```
0200  20 86 8B  GKEY    JSR ACCESS     ; Unprotect SYSRAM
0203  20 88 81          JSR SAVER      ; Save registers
0206  A9 00             LDA #00        ; Define User Port A
0308  8D 03 A8          STA A803       ; as an input
020B  AD 01 A8          LDA A801       ; Get ASC1l if any
020E  F0 13             BEQ GOBAK      ; If none, exit
020F  48                PHA            ; Stack it
0210  20 72 89          JSR BEEP       ; Tell user key found
0213  20 24 02  TSTA    JSR KYSTAT     ; Key still down?
0216  D0 FB             BNE TSTA       ; Wait in loop
0218  20 9B 89          JSR NOBEEP     ; Debounce delay
021B  20 24 02          JSR KYSTAT     ; Still down?
021E  D0 F3             BNE TSTA       ; Keep waiting
0220  68                PLA            ; Bring ASCII back to A
0221  4C B8 81  GOBAK   JMP RESXAF     ; Restore reg except A, RTS.
0224  48        KYSTAT  PHA            ; Save A
0225  A9 00             LDA #00        ; Define user Port A
0227  8D 03 A8          STA A803       ; as input
022A  AD 01 A8          LDA A801       ; Get ASCII if any
022D  0A                ASLA           ; Shift keystrobe into Carry
022E  68                PLA            ; Restore A (no effect on carry)
022F  60                RTS            ; Return
```

This routine above takes fewer bytes than the original and uses no zero page locations other than those ordinarily used by the monitor. To initialize the monitor to use the ASCII keyboard, we'll use the following INIT routine.

```
0230 20 86 8B INIT   JSR ACCESS     ; Unprotect SYSRAM
0233 A9 00           LDA #L,GKEY    ; Modify keyboard INVEC
0235 8D 61 46        STA $A661
0238 A9 02           LDA #H,GKEY
023A 8D 62 46        STA $A662
023D A9 24           LDA #L, KYSTAT ; Modify status vector
023F 8D 67 A6        STA $A667
0242 A9 02           LDA #H,KYSTAT
0244 8D 68 A6        STA $A668
0247 4C 03 80        JMP WARM       ; Warm Entry-Monitor
```

A quick refresher for those who may not have access to the February, 1980 MICRO: The ASCII keyboard is connected to user input Port A device U28 at connector AA with pin assignments as noted below:

ASCII Keyboard Connections:	0	1	2	3	4	5	6	Any key-down (+) strobe	GND
AA-Connector Pins :	D	3	C	12	N	11	M	10	1

To relocate the GKEY routine, modify the address at 214, 215 and 21C, 21D to point to the relocated KYSTAT routine. The INIT routine will also have to be modified to point to the new locations occupied by GKEY and KYSTAT. For those of you not familiar with the RAE (SYM Resident Assembler-editor) format, the notation in INIT showing LDA #H,KYSTAT or #L,KYSTAT means to take as an immediate value the High or Low byte of the address KYSTAT.

The routine could also be generalized to accept any number of keyboards at any number of input ports by modifying those locations showing addresses A801 and A803. For example, we could modify the SYM to add extra input ports as outlined in the SYM Hardware Theory of Operations Manual, and change the instructions at 208 and 227 to STA A002,Y (99 02 A0) and instructions at 20B and 22A to LDA A000,Y (B9 00 A0).

Then before calling GKEY or KYSTAT, you would load Y with the value representing which I/O port is to be accessed (hex 00 = Port B on I/O chip 1; hex 01 = Port A on I/O chip 1, hex 10 = Port B on I/O chip 2; hex 11 = Port A on I/O chip 2 and so forth). This change will make the routine fully relocatable, eligible for ROM use and useful for more than one I/O chip address (good for 16 chips comprising 32 ASCII keyboards if desired). There are other, more efficient ways of handling multiple input devices on minimal port assignments, but that's a separate subject This just presents one possibility.

The Reset Sequence

Now that we've worked out the major part of the monitor subroutines, (except for tape control routines which come later), it's time to see what we need to set up the monitor communications in the first place.

The reset function is fully described, from a hardware standpoint, in my Hardware Theory Manual, however, as part of the reset function, the software sequences are described in the following paragrahs.

The reset pulse originates with the onboard 555 timer. It is triggered either by the onboard reset switch or by power-on of the circuitry. (A characteristic of the 555 timer is that in one-shot mode it will produce an output pulse immediately on power-up, then it will wait for the trigger input to produce any other pulse output).

The first reset function is to define the stack pointer contents as hex FF. Next a hex CC is stored in PCR1 which is a control register on the 6532. This disables the power-on reset (POR) and turns off the tape motor.

Next, the interrupt (IRQ) is disabled by setting the 5th bit of the flag register. Since we are unable to set the bit directly, the A register is loaded with a hex 4 then is pushed onto the stack. The stack contents are pulled from the stack, setting that bit.

Then the monitor removes the write protection from the system RAM by the JSR ACCESS at 8B56. This prepares the RAM from A620 through A67F to be initialized to all of the "default" monitor-established values. These values are copied directly from the monitor ROM, locations 8FA0-8FFF by using the index register as a loop counter and absolute-X-indexed addressing for the load and store (loop from 8B59 through 8B63).

The next step is to tell the user that the monitor RAM is initialized. This is accomplished by the LDA #7 and JSR OUTCHR at 8B66. If you go back to 89C6, it does a compare to a value 07, and if true, sounds the onboard keypad.

KSCONF is called to determine whether the onboard keypad is to be used or an external CRT terminal. The monitor will remain in a loop from 8B6C through 8B75 testing first the keypad, then the CRT terminal port for an active bit. If there is a keypad press, it jumps to MONENT (8B7C).

The key keypad addresses are already copied into the input and output vectors, so we don't have any changes to make here. MONENT, then, only needs to restore the stack value to FF, unprotect the system RAM, then go to the warm entry point for the monitor. Since this is the basic command entry loop, we're now set to receive commands from the keypad.

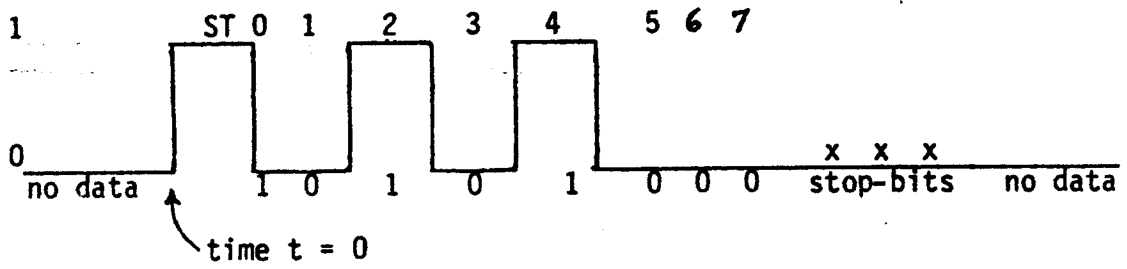
CRT I/O-

If we're running a CRT terminal, the SWLP loop will fall through the 8B76 where the input and output vectors are changed using subroutine VECSW. From then on, each INSTAT, INCHR or OUTCHR subroutine call will go to the terminal I/O port rather than the onboard keypad routines.

Since the CRT terminal is a serial device, we'll have to find out how fast the bits are coming at us from the terminal. Subroutine BAUD is called for this.

Before discussing BAUD, we must first understand the sequence of inputs which wll be seen on the input port when we try to log-on with the CRT terminal. The instructions indicate that a "Q" is used.

Remember that for a CRT terminal a no-data signal is seen as a continuous 0 level. A start bit is a one level and each real 1-bit is a 1 level. Note that bits are sent LSB first. Therefore a "Q" will look like this: (just the opposite of the TTY format).



To find the correct baud rate, we start with Y equal to zero (8B01). Then SEEK the first zero transition (stay in a loop 8B02 through 8B07 until bit 7 of port at A402 first goes low).

Now we can begin timing; use JSR INK to increment Y, delay for a certain time, then check if the input is still low. Stay in a loop counting the delay periods until input goes high. Do the same for one period where the input is high.

Then we compare the number found to the count which represents the nearest standard value, and store the standard value at SDBYTE. From here on, SDBYTE will establish our time delays for terminal I/O.

After the baud rate is established, we get to MONENT (8B7C) just as before and enter the main monitor loop.

TAPE I/O ROUTINES-

There are two primary entry points to the tape I/O section of the monitor. These are DUMPT at 8E87 and LOADT at 8C78. These are to dump memory to tape or load memory from tape respectively. As we did before with the paper tape routines, we'll start with an examination of the dump routine.

Rather than attempt a deep analysis of these routines, the primary reference to the understanding of these sequences would be the Appendix C of the SYM Reference Manual. Specifically, the entire sequence of the tape subroutines is structured to output the precise order of characters indicated in each of the tape formats. With this appendix as a reference, we can begin a walk through the dump routine.

At 8E87, subroutine START (8DA9) is called. This stores the Y register in location FD (called "mode"), unprotects the SYSRAM, sets up for tape output, and calls ZERCK to zero the checksum. Then it moves the start address of the dump from parameter 2 to 00FE, 00FF and turns on the tape recorder.

At 8E8A, port B of the 6532 is loaded with the number 7. This had the effect of applying 3.2 Volts (TTL high) across the audio output voltage divider network R88, R89, R90. From this point onward, this particular bit (pin 9 of U37) is manipulated to put the audio out to the tape recorder.

The routines from 8E8F thru 8EF9 are configured to output the control and data bytes in the order shown in Appendix C of the Reference Manual.

You will notice that, as with all of the other monitor routines, 00FE and 00FF are used to hold a critical value. In this case it is the current address of the byte being dumped (see 8EFC). After each byte, the contents of 00FE and 00FF are compared to the ending address low and high to see if the last byte has been sent out. (8ECE-8EDB). If not we stay on the loop until the last byte is gone.

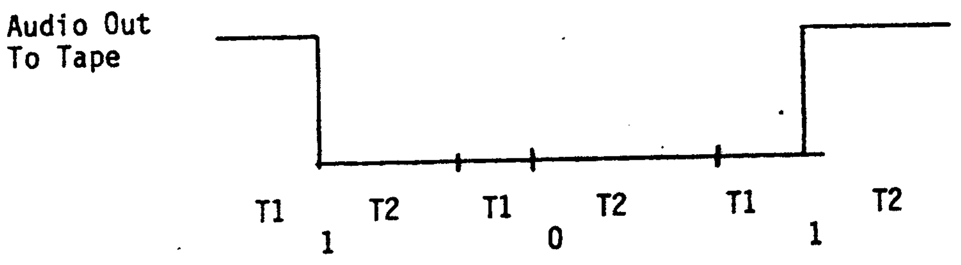
If we're done, the slash (end of record, hex 2F) is written, then the checksum (2 bytes), then tow EOT characters (hex 04). To exit, we go to 8D41, clear carry, branch from 8D44 to 8D4E thru to 8DBB which turns off the tape motor. Then return to the calling routine.

The output routine itself is located between 8F0E and 8F38. Wht is basically done here is to take the bits of the output byte, one at a time, and shift them left thru the carry bit. For every zero which enters the carry bit, there will be no change in the output level. For every one-bit which enters the carry bit, the output state of the tape audio bit will change from a high to low or vice versa.

The tape output for each bit is done in two segments. The first is controlled by the loop from 8F1F to 8F24. The loop constant is stored at TAPET1 (A635). The default value is 33. The second loop is from 8F2C thru 8F31. The delay constant is at TAPET2 (A63C), default value is 5A.

What actually goes on the tape, then, is a non-symmetric waveform as shown in the figure below. This waveform should aid in understanding how the readback circuit can distinguish between a one and a zero when we try to reload the tape.

A SAMPLE SYM-FORMAT TAPE WAVEFORM
(only 3 bits shown for clarity)



Because of the time constants used, hex 33 and hex 5A, the waveform for a one spends roughly 1/3 of its time (first segment in one state (hi or lo) then 2/3 of its time (second segment) in the other state.

When we are trying to reload the waveform, we set the high speed tape boundary at 46 hex. This is half way between the two values 33 and 5A.

At 8D66, during readback is where the boundary is set up. The RDBYTH routine at 8DE5 in conjunction with GETTR at 8DCA samples the input waveform from the signal conditioning circuitry and looks for a level transition occurring within the time representing 50 percent of a full expected timing cycle.

If there is a transition from a 1 to 0 or 0 to 1 ,the sensed bit is a 1, just as it was recorded in the first place.
If there was no transition within that time, thie bit sensed is a zero.

The rest of the read-tape routines follow along to load the data in the same format in which it was dumped. Again note that 00FE and 00FF are used to hold the address of the data as it is being read in from the tape.

Because the high speed values are stored in the RAM, we would be able to change them if we wanted to do so. What would happen then is an increase or decrease in the packing density of the data on the tape with a correspond decrease or increase in the speed with which programs may be loaded or store

Depending on the quality of your tape and tape recorder, you may be able to load or store programs up to 4 times faster. Try, for example, storing 1A at A635, 2D at A63C and 23 at A632. Now try saving a tape file, then reloading it. It should save and load roughly twice as fast as before. Note that tape speed variations will be twice as critical now and the level controls may need to be adjusted more cirtically.

Fro further data on modifications to the tape circuitry to accommodate the higher speed, please refer the SYM-PHYSIS, the SYM users-group newsletter.

This monitor circuit is provided also with the ability to save and load KIM format tapes. As with the high speed SYM format, the load and save routines are directed towards the specific tape format shown in Appendix C of the SYM Reference Manual.

As you will notice in the section from 8F56 to 8F97, the KIM character store routine is very solidly controlled as to its timing (all constants in ROM used in the timing loops). We could not change any of them as we did for the SYM format dump routine.

However, if KIM-format tapes were dumped in a higher speed fashion, such as by KIM Hypertape or Ultratape, maybe by changing (decreasing) the value at A631 (KMBDRY) one might be able to read these other formats successfully. SYM-PHYSIS will soon be reporting on the success of such attempts and this manual will be revised accordingly to report their results.

USE OF THE MONITOR SUBROUTINES-

The major portion of the monitor subroutines are shown in chapter 9 of the SYM Reference Manual. Because of the use it has been given in this book, NOBEEP (899B) should probably be mentioned here as a short term delay routine (same duration as a BEEP routine but no output). You'll see it used again later.

For the rest of the monitor routines, as a general note the ACCESS routine should be called in a users program (once at the beginning is usually enough), before you try to use the monitor routines. This is because many of them write into the SYSRAM as input buffers, parameter storage, scope buffers, etc. This call to ACCESS is a part of my own general policy for that reason. I had considered removing the write protect monitor jumper (MM-45) but I believe this facility may come in handy in the future.

Rather than go into further detail on the use of each of the monitor routines, having already explained the individual operation of each, I'll refer you to Chapter 9 of the SYM reference manual which has a summary of each. I will, however, be happy to answer any questions you may have about them. Since we have already traced through each of the routines by means of the flow narratives, the charts in chapter 9 of the reference manual can serve as a useful guide for the use of most of the routines.

USE OF THE MONITOR ITSELF AS A SUBROUTINE

Lets say we had a need during our program to fill an area of memory with a specific byte; to make a movement of a whole block of memory or to load a specific program from the tape.

Much effort could have been dedicated here to tell you exactly what parameters to set up and what new entry points would be used for each monitor function as individually called. However you would then have perhaps 25 different routines to look up and load, one for each monitor function you wanted to use.

Fortunately there is one simple routine I've developed which allows you to call any of the monitor functions except JUMP or GO by a special use of the EXEC function. (The JUMP and GO are restricted as explained below.) The best part about this routine is that is may be called, as a subroutine, from a machine language program or BASIC or most any other program language. Here it is:

SUBROUTINE EXECLNK - CALLING THE MONITOR FUNCTIONS

```
1000   20 86 8B   EXECLNK  JSR   ACCESS    ;access SYSRAM
1003   20 88 81            JSR   SAVER     ;save registers
1006   A9 04               LDA   #04       ;disable IRQ
1008   48                  PHA
1009   28                  PLP
100A   A9 10               LDA   #$10      ;point to area
100C   8D 4B A6            STA   $A64B     ;where ASCII-coded
100F   A9 50               LDA   #$50      ;monitor commands
1011   8D 4A A6            STA   $A64A     ;are stored
1014   20 55 88            JSR   $8855     ;change INVEC to RIN
1017   A9 1B               LDA   #$1B      ;ESC char to KTM/2
1019   20 47 8A            JSR   OUTCHR    ;sent it out
101C   A9 45               LDA   #'E       ;clear screen char
101E   20 47 8A            JSR   OUTCHR    ;sent to KTM
1021   20 9b 89            JSR   NOBEEP    ;delay while it clears
1024   20 35 80            JSR   USRENT    ;enter monitor
1027   20 86 8B            JSR   ACCESS    ;(sometimes returns protected)
102A   AD 3A A6            LDA   $A63A     ;restore
102D   8D 61 A6            STA   $A661     ;invec to
1030   AD 3B A6            LDA   $A63B     ;same as
```

```
1033    8D 62 A6        STA   $A662    ;before
1036    08              PHP            ;reset interrupt
1037    68              PLA            ;bit to allow
1038    29 FB           AND   #$FB     ;interrupts
103A    48              PHA            ;again (set I=0)
103B    28              PLP
103C    4C B8 81        JSR   RESXAF   ;(or RESXF or RESALL)
```

Lets look at this routine in detail.

At 1000, we allow access to the System RAM. This is done so that we can rewrite some of the input vectors. Then at 1003, we save the registers with SAVER since some of the registers may be in use by the calling routine.

At 1006-1009 the I (interrupt disable) bit is set to prevent IRQ's from messing up the program sequence. This is purely optional and, along with the instructions from 1036-103B, (which restore the I bit to zero), could be deleted at the users option.

At 100A, parameter 3 is set up with the address in memory where we have stored a block of ASCII encoded monitor commands. These are stored in the same format expected by the "EXEC" command function.

A special note must be made of the restrictions on the command block contents. Specifically, on entry to this routine, at 1014 we will be changing the input vector, storing the old input vector in monitor RAM scratchpad locations A63A and A63B. We also will have stored, on the stack, the data, flags, and return addresses from the previous levels of subroutine calls.

If we are ever to get back to the calling routine, we cannot do anything to disturb the previous contents of the stack. Both JUMP instruction and the <u>one</u> parameter GO function set the stack pointer to $FF again, then push on th address 7FFF representing (after autoincrement by 1) a return to the monitor after the GO or JUMP routine is completed if an RTS is at the end of the program sequence.

As a result of this reconfiguring of the stack, to use this routine we cannot allow the stack to be disturbed this way. So we will not use either of these two types of commands within our execute block for this routine.

At 1017, we've returned from changing the input vector. Now the next time the GETKEY routines are used, the monitor will be ready to find the input in the RAM instead of from the keyboard. (Keyboard will appear to be dead until the input vector is somehow restored.)

The sequence from 1017-1023 is written for the KTM-2 terminal. It is a screen clear sequnce which includes a delay while the KTM-2's processor is clearing its RAM to blanks. (Without the delay we generate some garbage on the screen when the next output occurs.) Again, this sequence is optional.

At 1024, we enter the monitor program at the USRENT (8035) point This follows a pseudo-interrupt sequence which means that, as in IRQ or BRK interrupts, the registers are saved within the SYSRAM in A65B-A65F and the program counter contents in A659 and A65A. This is done so that the register contents at the time of the interrupt may be displayed and to provide a place from which they may be restored to their original value to continue program execution when the interrupt sequence is completed. Further information about interrupt handling may be found elsewhere in this manual.

Now that we've entered the monitor at $8035, it will perform various setup tasks, then it will do a warm start to the main monitor flow. This folllows the sequence - get command, dispatch to execute it, write error msg if appropriate, recycle to get command again.

Since we modified INVEC to point to RIN, (see code from 8855-887D), it means that all commands will be taken in ASCII from that point in RAM specified in bytes stored at 00FE and 00FF. Just to supply an example, lets suppose we wanted to Fill AA-800-8FF and Block Move 900-800-8FF then to Verify 800-9FF and return control to the calling program. A sample program sequence to do this follows next.

SAMPLE SEQUENCE USING EXECLNK

```
          THE PROGRAM:   0200 20 00 10       JSR  EXECLNK

                         0203 00             BRK

          THE DATA:      1050     46 41 41 2C 38 30 30 2C 38 46 46 0D
                                  (F  A  A  -  8  0  0  -  8  F  F  cr)

                         105C     42 39 30 30 2C 38 30 30 2C 38 46 46 0D
                                  (B  9  0  0  -  8  0  0  -  8  F  F  cr)

                         1069     56 38 30 30 2C 39 46 46 0D
                                  (V  8  0  0  -  9  F  F  cr)

                         1072     47 0D      (For this EXECLNK, always the
                                  (G  cr)     last command in the sequence)
```

You will notice at the end of this data representing the execute block, instead of a "00" which would normally represent the end of the block (refer to EXEC command description), we have used a "GO CR" instead.

This is a zero parameter GO command. As detailed earlier, this form of the GO will not reinitialize the stack, but will only resume the execution of the program at the location referenced by PCHR, PCLR (A659,A65A) after restoring all of the registers to their original contents at the time of the interrupt.

Since the JSR to USRENT was treated as a pseudo-interrupt, it means we'll be going back to location 1027 to find our next instruction (the first location after the interrupt).

At 1027 we again must allow writing to SYSRAM because certain of the monitor routines could cause it to be protected. We are going to need access again so that we can restore the input vectors which are contained in the SYSRAM

Next, the code from 102A-1035 performs the same function as RESTIV (8899-88A4) but we've provided the extra capabilities here to restore the interrupt capacity (1036-103B) and to pass variables in A or F by the jump to RESXAF.

For those of you who are using SYM BASIC, this routine may be called from BASIC just as easily as from a machine language routine. A sample is shown below for your reference. To conserve memory, you'll probably want to move it somewhere else. For the move, we'll discuss changes after this example.

CALLING EXECLNK FROM BASIC (Assumes an 8K SYM)

Store the routine in 1000-103E, data in 1050-1073. Then cold start BASIC as follows:

```
Memory Size? 3500 (cr)

Width? (cr)

10 PRINT "CALLING EXECLNK ROUTINE"

20 X=USR(&"1000",0)

30 PRINT "RETURNED OK"

40 END
```

If you RUN this routine, you'll see exactly the same result as before but watch out when you're using monitor routines at the data move level, they could clobber your BASIC text programs.

Lets talk about generalizing the program so that we could put it into ROM if we wanted to. First, the routine itself was written to be fully relocatable. This means we could decide to put it anywhere in our memory space and it would work the same way without any changes. But the execute block area would be fixed at $1050 so we'll have to do something about that.

We have decided not to use the JUMP function while this routine is active, so we have a few monitor RAM locations available to us.

Lets use A626-A62A for our data. Our changes to the program will then be as follows:

```
        At 100A, instead of A9 10
                 we'll use  AD 27 A6
                 (pushes everything else forward one location)
        At 100F (now 1010) instead of A9 50
                           we'll use  AD 26 A6
                 (pushes everything else forward one location)
        At 103C (now 103E) instead of 4C B8 81
                           we'll use  20 28 A6 4C B8 81
```

Now the final routine can be located in ROM anywhere in the memory space. To use it, we can place the following routine in RAM and execute it once. This routine sets up the proper addresses of the execute block and for the ending routine to be executed before returning to the main calling program.

To explain this further, in order to fully generalize the ROM-able version of the EXECLNK routine, I have inserted a JSR $A628 just prior to executing the JMP RESXAF at the end of the routine. This allows the user to place the address of a routine which he wishes to execute after the exec block is done and just before returning to the calling program. The users code will end with a 60 (RTS) which will allow for a normal return.

This address vector is set up by the code in 070D-071B in the example which follows. The other function of this sample setup routine is (0700-070C) to tell the EXECLNK subroutine the beginning address of the ASCII encoded execute block of commands.

So just to repeat, to use the EXECLNK routine, store away your exec block somewhere, then execute a routine similar to this one to tell the monitor where this execblock is located and the address of the routine you want executed before returning to the calling program. Here's the sample setup routine.

```
0700  20 86 8B   SETUP   JSR  ACCESS
0703  A9 10              LDA  #$10    ;use your choice
0705  8D 27 A6           STA  $A627   ;of exec block area
0708  A9 50              LDA  #$50
070A  8D 26 A6           STA  $A626
070D  A9 4C              LDA  #$4C    ;setup a new vector
070F  8D 28 A6           STA  $A628   ;for a routine to do
0712  A9 07              LDA  #$07    ;before we return to
0714  8D 29 A6           STA  $A629   ;the calling routine
0717  A9 50              LDA  #$50    ;As an example, use
0719  8D 2A A6           STA  $A62A   ;a routine at $0750
071C  60                 RTS
```

A test program would then be as follows...

Assuming I had, as in the previous example, stored my exec block at 1050 and up, and had left the ROM-able version (modified as suggested) at location 1000 and up, we could do the following example. With the preceeding setups, start this routine at 0200.

```
0200  20 00 07  TEST  JSR  SETUP
0203  20 00 10        JSR  EXECLNK
0206  00              BRK

0750  A9 58   ENDRTN  LDA  #'X
0752  20 47 8A        JSR OUTCHR
0755  60              RTS
```

INTERRUPT HANDLING ROUTINES-

Earlier we had examined the effect that USRENT had in the process. Before examining that one in detail, we'll take a look at the general flow of interrupt handling. As this requires an understanding of the way the processor handles interrupts in the first place, we'll review that first.

All interrupts, when they occur, cause the processor to complete the instruction which it is executing, then it will push the contents of the program counter on the stack (HI byte then LO). Then the flag register is pushed onto the stack. Then the program counter is loaded with the address of the interrupt routine specific to that type of interrupt.

The addresses for the specific routines are stored in FFFE,FFFF (LO,HI) for IRQ or BRK interrupts and in FFFA,FFFB for NMI (nonmaskable) interrupts. Nonmaskable refers to a interrupt which will occur regardless of whether the I flag is 1 or 0. Any interrupt will set the I flag so that we will prevent any other interrupts from occuring if desired when we are busy processing the current one.

Since we said the NMI interrupt was the only one we could not stop with the I flag, lets look at this one first.

For a NMI interrupt, the address pulled out of FFFA and FFFB is 809E. It accesses the SYSRAM and sets the carry bit. Then it calls SAVINT.

As of the entry to SAVINT, the stack's most recently added items are the (return address-1) to the calling routine at 80A5, and the flags-at-interrupt and the PCH, PCR-at-interrupt.

As we had done with SAVER, lets look at the stack relative to the indexed-referenced locations. In the table which follows, S represents the stack value just before the NMI occurs.

Indexed Stack Data Location

Stack Value	Data Location	Current Contents	Contents after SAVINT, before its RTS
S	105,X	PCH	Return Address Hi byte
S-1	104,X	PCL	Return Address Lo - 1
S-2	103,X	Flags	Don't care
S-3	102,X	Rtn addr Hi	Don't care
S-4	101,X	Rtn addr lo-1	Don't care
S-5	100,X		Stack value on entry to SAVINT

On entry to SAVINT, the A, X, and Y resisters are saved in their respective areas AR, XR, YR (A65D, A65E, and A65F). Then, as with SAVER, we transfer the Stack register to the index X register so that we know where specific data can be found on the stack. Decimal mode is cleared to prevent any difficulty with the arithmetic at 8072 and 807A.

Then we pick up the PCL value from the stack (S-5+104 = S-1). We then add $FF <u>plus</u> carry (recall that the carry bit was set before entry) with a result of a apparent addition of zero with a preset carry bit for the next operation (this is significant for USRENT as described later.) Then we store the result in PCLR (A659). We now pick up the PCH from "105,X" adjust the value and store it at PCHR (A65A) and the flags from "103,X" and store them at FR (A65C).

Since the flags and program counter are already put away for future use, we no longer need to keep them on the stack. Therefore we can reshuffle the stack as shown in the rightmost column of the above diagram. The stack pointer is made "S-2" by the bytes at 8091 to 8094.

The last action of SAVINT is to increment the X register twice and store its value at SR (A65B). This establishes the value the stack had at the time the interrupt occurred. Then we return to the calling routine.

At 80A2, we call DBOFF (80D3) which disables the NMI interrupt (note that we can disable it with hardware rather than with the I flag). This is to stop any attempt to service a second NMI while the first one is being serviced. Then we check TV (Trace Velocity) at A656. If it is zero, we just jump to the display routine (PCH, PCL, plus a "2" are displayed to indicate an NMI). If TV is not zero, we are involved in an instruction trace (with delay). Since the trace function is covered in the SYM Reference Manual, the description will not be repeated here.

The fineal interrupt service activity is a warm start to the monitor from 80BD. This puts it into command entry mode.

Processing of the IRQ or BRK interrupts begin with the same routine. From address FFFE and FFFF we get 800F as the starting point whether IRQ or BRK interrupts have occurred. You'll notice there is only one address pair for handling both types of interrupts. It is up to us to determine which type of interrput had occurred and process them differently if we wish. The code from 800F-8018 sets up to discover if the BRK flag is on and allows for the transfer to the appropriate service routine.

Due to the use of interrupt vectors at 801F and 8026, the user can change these vectors to establish his own IRQ or BRK service routines. These vectors, stored at FFF6, FFF7, or FFF8, FFF9 can be changed by calling ACCESS then storing new vectors at A676, A677 (FFF6,7) or A678, A679 (FFF8,9) due to the mul;tiple address decoding of this area. See my Hardware Theory of Operations Manual for further details on incomplete address decoding.

The last of the interrupt handlers is USRENT, in actuality a pseudo-interrupt generator. A jump to subroutine (JSR) to USRENT (8035) causes the monitor to pretend an interrupt has occurred and to adjust everything accordingly so as to process it in the same way.

Since we entered at 8035 by means of the JSR, it means that the return address - 1 will be located on the stack. This stack contents is not what is expected by an interrupt processing routine. If you refer to the 6502 Programming Manual, you'll find that during entry to the interrupt, the processor has pushed the program counter contents (actual return address, not return address - 1 as done with a JSR) onto the stack, then the flag register contents.

Since a JSR does not push the flag contents onto the stack, if we are to handle this in the same manner as other interrupts, we must format the stack in the same way and adjust the return address to reflect the correct return. At 8035, therefore, we push the flags onto the stack, unprotect the SYSRAM, set the carry flag, and call SAVINT.

On our return, we increment PCLR (and PCHR if a carry resulted from the PCLR adjust) which has the effect of correcting the program counter contents for a correct return. Now everything appears as though a true interrupt has occurred.

SYM REFERENCE MANUAL CORRECTIONS-

As an aid to the understanding of the routines presented in this manual, it was necessary to examine the SYM Reference Manual rather closely. I have therefore provided here a small set of corrections to errors typeset in the June, 1979 printing as certain of them tend to cause some difficulty.

Page C-2, 4th waveform is a "Squared 0"

Page H-1, KEYQ returns Z flag=0 (result is not zero) if any
key is pressed; same error at program lines 1195, 1223.

Location 8493: comment field should say "Mode = KIM"

BIBLIOGRAPHY

1) SYM/KIM Appendix to the First Book of Kim; Robert A. Peck, 1979

2) SYM-1 Hardware Theory of Operations Manual; Robert A. Peck, 1979

3) Expanding the SYM, Adding an ASCII Keyboard; MICRO Magazine, No. 21 p. 5 plus corrections MICRO Magazine, No. 24 p 39, Robert A. Peck

4) Staged Loading Techniques for Segmented Programs; MICRO Magazine, No. 20, page 59, Robert A. Peck

5) 6532 Timer ; MICRO Magazine, No. 17 page 55, Robert A. Peck

6) SYM Reference Manual; Synertek Systems Corp Third Printing, June, 1979.

7) SYM Programming Manual; Synertek Systems Corp 1978

8) SYM Hardware Manual; Synertek Systems Corp, 1978

9) 6502 Assembly Language Programming; Lance Levanthal, Osborne/McGraw Hill, 1979.

10) The First Book of KIM; Jim Butterfield, Hayden Books

Notes: A) Items 1, 2, 6, and 10 are available directly from the me. Write for price and delivery details.

B) The monitor, as noted, allows for easy user expansion. One such expansion package I have seen and used is by Jeff Holtzman. It includes a disassembler, software settable breakpoints and many other useful features. You may contact him directly regarding price etc at: 6820 Delmar - 203 St. Louis, Mo. 63130