PORT FE
PORT FE
PORT FE

Sorcerer Users' Group (Toronto) Newsletter
200 Balsam Ave., Toronto, Ont., M4E 3C3
Volume 1 Number 5
July 1980

## EDITOR'S TURN

This month I have not written a critical editorial essay for several reasons, the first being that I haven't had time to prepare anything. Actually, I did have something small prepared and I had 75% of this newsletter typed into Duncan's computer. But some stray electrical charge wandered along (its not the first time on this system) and blew the whole thing. I absolutely refuse to type the whole thing in again and anyhow, it wasn't very interesting. You're not missing anything. However, next month I plan on doing an article which discusses the value and uses of several different programming languages. I state quite clearly, though, that I am not an expert on all programming languages nor that I use all of them extensively, if at all. They tend to fall into classes of style and usage, however, and can be discussed from that viewpoint.

I would again like to urge everybody to submit articles for the HARDWARE and the EQUIPMENT REVIEW columns. I would like to see some articles on minor modifications to the Sorcerer or peripheral equipment which is easy to perform and is interesting and/or useful. I would also like to urge everyone to make whatever donations you possibly can to the software library. There are currently many competitions going on with some of the games from the library and it all seems to be working out great.

## SOFTWARE

Over the last few months, I have been showing how to do some very interesting and useful I/O routines for both the keyboard input and video output. This month, I will conclude that series with some programmes that allow you to test for keys using the keyboard I/O port.

All I/O on the Sorcerer is done using port FE (255 decimal). By testing the particular values on that port, it can be determined which key on the Sorcerer keyboard is physically being pressed. This is not the same as testing for its ASCII value. The ASCII values that are assigned to keys are arbitrarily set by the keyboard input logic routines in the EXIDY monitor ROM. There are always two ways of handling I/O in any system - the physical and the logical. They are very different. When pressing a key on the keyboard, the monitor routines test for physical occurrences which are determined by the changing values (bit patterns) on the ports and then logically assign some standardized value to them (for meanings' sake) such as an ASCII number. However, you don't have use that logic if you don't want to. The logic used in the monitor ROM is very well written and imperative for the general usage of the machine but it is not necessarily so for many specific applications, such as when speed or constant timing is critical. In these cases, you can test for physical occurrences yourself and write your own logic to suit your own needs.

In order to understand the physical keyboard set-up a little better, turn to the keyboard schematic in the Sorcerer Technical Manual (if you have one). If you don't have a technical manual, refer to Table 1 in this newsletter. The keyboard is a matrix or grid system which is tied into the input and output lines of port FE. It is a 16 x 5 = 80 key grid which uses the 5 lower bits of each of the input and output sections of the port. When you send a particular value on the output section of the port, it turns on one of sixteen lines. This line is connected to 5 keys on the Sorcerer. When you press one of these 5 keys, it completes an electrical connection which is detected by the input section of the port and can determine specifically if a particular key is on (pressed) or off (not pressed).

The output section operates a little differently from the input section. The line of 5 keys you want to test for goes out as a value which is one of the line numbers (0-15). (Note: the numbers listed beside the lines of the output chip in the schematic are the pinout numbers, not the line numbers.) Thus, the BASIC command OUT 254,0 would activate line 0 of port FE and OUT 254,8 would activate line 8 of port FE. The output section, then, is value patterned. But the input section is bit patterned. That is, one of the keys will turn on one of the 5 bits, 0-4. A very important note to remember is that the Sorcerer has inverse logic, so that an "on" bit is a zero and an "off" bit is a one. Therefore, if no keys are pressed, then all 5 bits will be ones. If one of the keys is pressed, it will appear as a zero in that key position. The values for any one of the 5 keys on any line will then be:

| Binary | Hex | Decimal |
|---|---|---|
| 11110 | 1E | 30 |
| 11101 | 1D | 29 |
| 11011 | 1B | 27 |
| 10111 | 17 | 23 |
| 01111 | 0F | 15 |

Now we can make a test to see if a particular key is being pressed. Let's test for the RUN/STOP key. From the schematic or Table 1, we can see that the RUN/STOP is on output line 0 and on input bit 0. I will write this from now on as KEY(out,in) making the RUN/STOP key printed as RUN/STOP(0,0). In Z-80 this would look like:

```
TEST    XOR  A          ;make Acc.=0
        OUT  (0FE),A    ;turn on line 0 of port FE
        IN   A,(0FE)    ;get input from port FE
        BIT  0,A        ;test bit 0 of the acc.
        JP   Z,R/S      ;go if RUN/STOP pressed
NO$R/S  ;service routine for case of
        ;RUN/STOP not pressed
R/S     ;service routine for case of
        ;RUN/STOP pressed.
```

Now, last month I said that this could be done with ROMPAC BASIC using the OUT and INP commands, but that was a mistake. It cannot be done from BASIC for this one simple but irritating reason. BASIC takes the opportunity between statements to do a quick check of the keyboard to see if the user wants to abort the programme. The aborting keys reside on line 0 and, therefore, between the OUT and the INP commands, the BASIC PAC always resets the output section of port FE to 0 which makes the test useless for all but the test for line 0 keys. A machine language subroutine will then have to be set up to handle this. We'll make this routine flexible so that it can easily be changed from BASIC to test any bit on any line. We'll start the routine at address 0:

```
Add    Code      Label      Mne   Oper       Remarks

00     21 20 00  TEST       LD    HL,0020    ;FLAG STGE ADD.
03     36 00                LD    (HL),0     ;CLEAR FLAG
05     3E 00                LD    A,LL       ;LL=LINE #
07     D3 FE                OUT   (0FE),A    ;TURN ON LINE LL
09     DB FE                IN    A,(0FE)    ;GET INPUT BITS
0B     CB 00                BIT   BB,A       ;BB=BIT #
0D     C0                   RET   NZ         ;NZ=NOT PRESSED
0E     36 01                LD    (HL),1     ;SET FLAG FOR PRESSED
10     C9                   RET              ;BACK TO BASIC
```

Now from BASIC we can first POKE the output section line number we want into location 6 and the proper code for the BIT test into location 12 (0CH), then call the subroutine using the USR function. When it returns, we can PEEK location 32 (20H) to see if the key was pressed. We can now write a general purpose BASIC subroutine to do this called SCAN. When we call this subroutine, we will pass it two arguments, LINE which will specify the line number 0-16 and BIT for the bit number 0-4.

```
XX00 REM: SCAN SUBROUTINE
XX10 RESTORE XX90:REM RESET DATA TO LINE XX90
XX15 REM GET BIT CODE FOR BIT POSITION
XX20 FOR I=0 TO BIT:READ CODE:NEXT I
XX25 REM NOW POKE LINE # INTO LOC. 6
XX30 POKE 6,LINE
XX35 REM AND BIT CODE INTO 12
XX40 POKE 12,CODE
XX45 REM NOW CALL M/L SUBROUTINE
XX50 POKE 260,0:POKE 261,0: M=USR(0)
XX55 REM CHECK FLAG STATUS
XX60 IF PEEK(32) THEN PRESSED=1:RETURN
XX65 REM ELSE KEY NOT PRESSED
XX70 PRESSED=0:RETURN
XX90 DATA 71,79,87,95,103
```

When this subroutine returns to the calling statements, if PRESSED=0 then the key was not pressed, else it was pressed. This routine now has a constant time limit which is the same for every call. Neither the processor nor the programme is dependant upon operator response time or the length of time that a key is kept down. You do not have to relocate and alter large blocks of code either. If you are using it for video games, then motion will be fluid across the screen. Another very useful function for this method of key testing is for a true randomization for Sorcerer. You can now actually measure the operator response time to use as a seed or you can put a RANDOM statement in a loop which continues to execute until the operator responds and releases the key. It would be virtually impossible to duplicate a response time at processor speeds.

Here's a routine in BASIC that will test the 4,5 and 6 key on the keypad that could be used to move a cannon left (4), right (6) or fire (5). For the sake of convenience and clarity, I will give the subroutines names instead of line numbers. SCAN is the testing routine given above, LEFT is a routine that points the cannon left, RIGHT points it right and FIRE fires the cannon. Of course you would need subroutines to count shots and score but these are left out here. Our routine is called CHECK.

```
X010 REM TEST THE 4 KEY
X011 LINE=13:BIT=2:GOSUB <SCAN>
X020 IF PRESSED GOSUB <LEFT>
X030 REM TEST THE 5 KEY
X031 LINE=14:BIT=2:GOSUB <SCAN>
X040 IF PRESSED GOSUB <FIRE>
X050 REM TEST THE 6 KEY
X051 LINE=14:BIT=3:GOSUB <SCAN>
X060 IF PRESSED GOSUB <RIGHT>
X070 REM ALL SCANNED SO DO AGAIN
X080 GOTO X010
```

I hope you have found these I/O routines useful. It is my feeling that with them, you can far surpass the I/O abilities and facilities of just about any other micro on the market. Start integrating some graphics routines with these and really open up some new fields for research and enjoyment.

BASIC is a fairly good language for small programme development on home and/or personal oriented microcomputers but it has some very severe limitations as we have seen over the last few months. It is not possible to gain a high degree of flexibility and control without introducing machine language subroutines. However, when machine language subroutines are integrated into the operating framework of BASIC, its basic power is greatly multiplied. Finding a comfortable and safe place to put these subroutines is another problem altogether.

Steve Dicker has submitted a very good article and programme on one method of dealing with this problem which is appended to this newsletter. I suggest it be read carefully to understand its full potential. Next month, I will show another method to handle this problem. With these routines and the I/O routines, there will be no reason why your programmes can't run as smoothly and look as professional (if not moreso) than that which comes out of large software houses.

## GENERAL NEWS

The next meeting will be held on THURSDAY JULY 10, 1980 at my home 200 BALSAM AVE. TORONTO at 7:30 pm.

By car:-
take the GARDINER EXPRESSWAY Eastbound to the end which turns into LAKESHORE BLVD. Continue on LAKESHORE to the end which turns into WOODBINE AVE. Turn right (EAST) at the first light which is QUEEN ST. Take QUEEN ST East about three or four stoplights to BEECH AVE. Turn left (north) on BEECH and go up the hill for 2 blocks which is the intersection of BEECH and BALSAM. My house is on the NORTHWEST CORNER of that intersection.

By TTC:-
take the SUBWAY to COXWELL STN and get the COXWELL SOUTH BUS. The bus will go down to QUEEN ST and come back up KINGSTON RD. Get off at BEECH and walk 1 block south to BALSAM AVE. My house is on the NORTHWEST corner of that intersection.

## TABLE 1

The following table lists the KEY(line,bit) in vertical groups of 5 from left to right across the keyboard. Note that the key symbol given is merely an identification of the physical key on the board and has no actual meaning other than that.

MAIN KEYBOARD

| | | | | |
|---|---|---|---|---|
| STOP(0,0) | GRAPH(0,1) | CTRL(0,2) | SHFT LOCK(0,3) | SHIFT(0,4) |
| CLEAR(1,0) | REPT(1,1) | SPACE(1,2) | TAB(1,3) | ESC(1,4) |
| X(2,0) | Z(2,1) | A(2,2) | Q(2,3) | 1(2,4) |
| C(3,0) | D(3,1) | S(3,2) | W(3,3) | 2(3,4) |
| F(4,0) | R(4,1) | E(4,2) | 4(4,3) | 3(4,4) |
| B(5,0) | V(5,1) | G(5,2) | T(5,3) | 5(5,4) |
| M(6,0) | N(6,1) | H(6,2) | Y(6,3) | 6(6,4) |
| K(7,0) | I(7,1) | J(7,2) | U(7,3) | 7(7,4) |
| ,(8,0) | L(8,1) | O(8,2) | 9(8,3) | 8(8,4) |
| /(9,0) | .(9,1) | ;(9,2) | P(9,3) | 0(9,4) |
| \(10,0) | @(10,1) | ](10,2) | [(10,3) | :(10,4) |
| RUB(11,0) | RETN(11,1) | LF(11,2) | ^(11,3) | -(11,4) |

KEYPAD

| | | | | |
|---|---|---|---|---|
| +(12,0) | *(12,1) | /(12,2) | -(12,3) | |
| 0(13,0) | 1(13,1) | 4(13,2) | 8(13,3) | 7(13,4) |
| .(14,0) | 2(14,1) | 5(14,2) | 6(14,3) | 9(14,4) |
| | | | =(15,3) | 3(15,4) |

# BASIC TO Z80 ASSEMBLY LANGUAGE INTERFACE

For those of you who have a Sorcerer Software manual, you already know that there are only two places to locate machine language subroutines which you intend on interfacing to a BASIC program. You can either place the subroutines in the first 256 bytes of memory or locate them in the BASIC free space at the end of your BASIC program. The first method places a size limitation on your subroutines while the second is risky because the subroutines may be overwritten when you are editing or updating your BASIC program. I recently discovered a third method of locating machine language subroutines which has none of the objectionable qualities of the two given above. You may allocate as much memory as you wish for your machine language subroutines and you can rest assured that they will remain safe through all your long hours of debugging the BASIC portion of your program.

Here's how you do it. First, decide how much space you are going to need for your machine language subroutines. Add the required number of bytes to 01D5H and add 1 to this sum. For example, if the subroutines required 200H bytes, your calculation would be:

```
  01D5H
+ 0200H
+    1
  -----
  03D6H
```

Now go into the monitor and ENTER location 149H. Type in the two byte address just calculated (remembering to reverse the order of the bytes). Using the above example:

```
    0149: D6 03 (CR)
```

Next, ENTER a 0 into location 3D5H. These numbers may also be POKED in using BASIC but I find it easier to do as described above.

Now return to BASIC and enter the command NEW. These actions reserve the desired memory space at locations 1D5H through 3D4H. If you now ask for the amount of FRE space you'll find that 513 bytes of memory appear to be missing. That is, a 16K machine which normally has 15592 bytes free after a NEW now has only 15079 bytes free. Your BASIC programs will now start at location 3D6H and will never walk over the reserved space.

Please note that the byte preceding the start of your BASIC programming area (at location 3D5H in this case) must be kept at a value of 0. If you start getting a SYNTAX ERROR message when you try to run your program it means that you have either forgotten to clear this byte or have overwritten it with your machine language subroutine. To get your program to run all you need do is clear this byte again.

When you want to save your program, you can save the machine language and BASIC portions together. Go back into the monitor and set locations 149H and 14AH back to their cold start values of D5 and 01 respectively. Return to BASIC and CSAVE your program. Everything from 1D5H to the end of your BASIC program will be saved as one unit.

To reload the program you must first go through the procedure outlined above for reserving memory and then merely CLOAD your program. If you forget to reserve the memory before loading the program then the BASIC will alter your machine language subroutine when it corrects the program link pointers after loading. It's a good idea to write down the address where each of your programs start when you save them in this way. Otherwise, unless you always reserve the same amount of memory, you won't know what address to load into 149H when you are reloading the program.

If you have a BASIC program that was originally written starting at 1D5H which you would like to alter to the format described above, you can do it as follows. Go through the previously described procedure of reserving memory (don't forget to zero the last byte of the reserved space!). Then CLOAD the program (using the full form of the CLOAD command) at the start of your BASIC space. Using the above example again and assuming you were loading a program called PRGM, you would enter:

```
CLOAD PRGM 1 3D6
```

Once the program is loaded, it may be saved as described above.

Well, that's it. If anybody tries this out and has any trouble or any interesting uses for it, I'd appreciate hearing from you. Steven Dicker