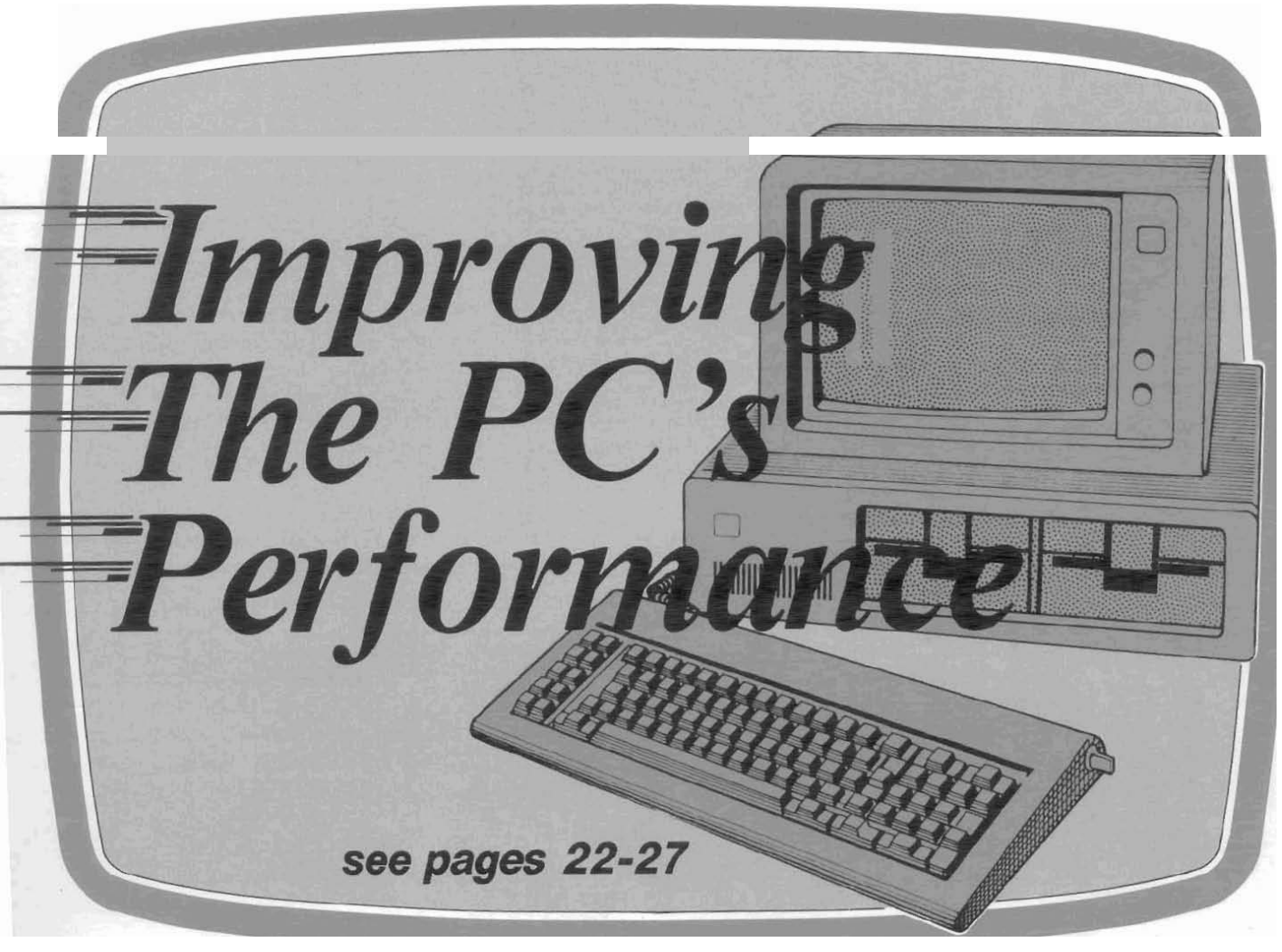
For the Advanced Computer User

Micro/Systems Journal™



Also in this Issue

Complete Table of Contents on Page 3

July/August 1986 Vol.2/No.4 $4.50 U.S.A.

Build A Smart Keyboard Interface

by John Monahan

Add Intelligence To Your Keyboard Input

Ever wished your keyboard had more keys or all your programs used the same control codes for cursor movements and editing functions?

This has always been my problem! Recently I decided to do something about it. After all, if I have my own personal computer, I should be able to have it do the things I want to do. I happen to have an S-100 system with Z80 and 8086 CPU's, and since I use an IBM-PC compatible keyboard and run MS-DOS, I wanted to run MS-DOS software that looks for an IBM keyboard.

My approach, which can be added to many other types of systems, is to have the keyboard talk to a keyboard controller which in turn talks to the computer. By having intelligence between a dumb keyboard and the computer, a truly powerful system can be set up, because the controller does not take up CPU space; it is always present, even on power up, works with different operating systems, and, as we shall see, can be much more powerful than its software counterpart.

THE CIRCUIT

To have any kind of flexibility, one must put together a unit containing an 8-bit microprocessor, a few I/O ports, as well as RAM and ROM memory. There are a number of very good single-chip, 8-bit microprocessors that have these features. The Intel 8048 is a classic example. However, in order to put something together quickly in hardware and to program it with an assembler I already had, I used the popular Z80. It requires few parts to assemble a very powerful system. A Z80 CPU with a 2716 ROM, 2Kx8 RAM, and two Zilog PIO's, together with a few small "glue" chips are all we need to construct a computer with 4 parallel ports with handshaking. Figure 1 shows the complete computer.

Because Z80 output pins are TTL compatible and capable of driving one TTL load, no buffers are required on the address, data or control lines. The 11 lines needed to address the 2716 ROM and 2Kx8 RAM chips can be connected directly to the Z80. These correspond to address lines A0 to A10. Address line A11 selects the memory in the ROM (A11 = 0) or RAM (A11 = 1). Pin 20 is the critical pin for selecting the 2716 ROM. Whenever this (CS*) pin is low, the ROM places data on output pins: 9, 10, 11, 13, 14, 15, 16, and 17. These are then read by the data-bus pins of the Z80 when the Z80 brings MEMR*,RD*, and A11 low. Any write-to-RAM-memory or R/W-to-an-I/O port will not cause the 2716 CS* to go low and so will not cause it to place data on the data bus.

The operation of the 2K RAM chip is a bit more complicated. We have two possibilities. First a read from the memory; Pin 21 (WE*) should be high. No problem here since it is connected to the Z80 WR* line. Pin 20 (OE*) of the RAM chip is connected to the Z80 RD* line. When this is low, along with the Z80 MREQ* and a high Z80 A11 line, the 2K RAM places the selected data (determined by address lines A0 - A10) on the data bus to be read by the Z80. For a write to RAM, memory-pin 20 (OE*) ofthe RAM must be held high. Pin 21 (WE*) is brought low by the Z80 WR* line. The CE* is selected as described above for the read cycle.

Because we are using only 8, of the possible 256 I/O ports available on the Z80, we do not have to decode the address lines completely to address the two Zilog PIO's. The PIO's are Zilogs' answer to the Intel 8255. They are 40-pin LSI chips that contain 2 separate 8-bit parallel ports with handshaking. They may be programmed in a number of configurations. More on this later.

The PIO's have a unique built-in capability to interrupt the Z80 in an ordered manner. We will not be using this characteristic of the PIO, however this requires that the Z80 clock and M1 pins to be directly connected (see Figure 1).

Each PIO is selected by bringing pin 4 (CS*) low. We do this on PIO1 and PIO2 by lowering address lines A2 and A3, respectively. Because each PIO contains 4 ports, we use address lines A0 and A1 to select these. Pin 5 of the PIO selects the command or data port. Pin 6 selects either port A or B on the chip. Address lines A0 to A3 will always be changing as the CPU reads memory, however, only when the IORQ* line goes low, will the PIO be addressed. This only happens when the Z80 code forces the CPU to read or write to a port.

The Z80 requires an external clock signal. Anything from 2 to 4Mhz is fine for this application. This is provided by the simple oscillator circuit connected to pin 6 of the Z80. Note that a 600Ω pullup resistor is required, since Zilog specifies the voltage swing must be within 0 to 5 volts.

CHECK OUT

So far we have assembled the bare essentials of a computer. With the appropriate software in ROM, the Z80, after power up, can be made to look at one data port and transfer the data across to another port. Complicated character translations can be done by adding on software as described below. If this is the first time you have put together a system of this complexity, you may first want to try something simplier first: namely fill a 2716 ROM with 76h's (the Z80 HALT instruction), switch on the power, and check that pin 18 of the Z80 has gone low, indicating the CPU has gone into the halt state. If this does not happen, more than likely you have one of your address or data lines connected incorrectly.

INTERFACING TO THE KEYBOARD

Connecting the Z80 board to an IBM-like keyboard entails one complication. The keyboard sends the data serially over two wires. One contains the data as 8-bits, the second, the keyboard clock data associated with the data. It would have been nice if IBM had chosen to use a standard UART-compatible, serial-data

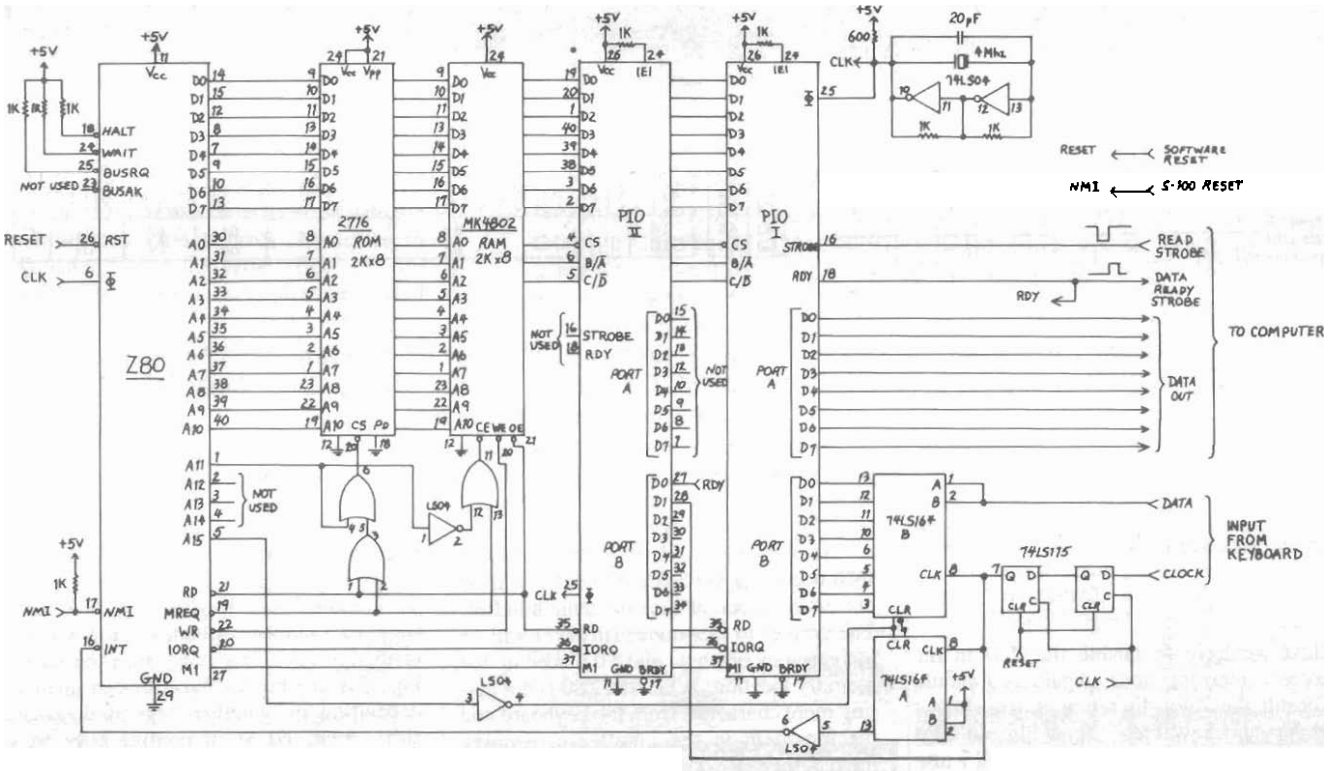


Figure 1. Keyboard controller circuitry

format. No such luck. The data is sent as 8-bits with no start bits or stop bits (Figure 2).

While it would be possible to program the Z80 to monitor one bit of an I/O port to assemble a byte from the serial data, the hardware solution of using two 8-bit shift registers makes life so simple that I opted for the easy way out. Here is how it works. Eight serial clock bits are shifted into the LS164-A-register (Figure 1) from the raw clock line coming from the keyboard. At the same time, 8 data bits are shifted into position in the LS164-B-register. Pins 3, 4, 5, 6, 10, 11, 12, and 13 of this chip will end up with the data in parallel form. The rising edge of the final clock bit raises pin 13 of the LS164-A-register. This, via the LS04 inverter, causes the INT* input to the Z80 to go low. This in turn causes the Z80 to call on an interrupt in ROM, which will pulse address line A15 high, clearing the two LS164 shift registers and readying them for the next byte from the keyboard. The raising of A15 is a cheap way of getting a fast 1-bit output port. Since our computer has no memory above 4K, we do not have to worry about the value of the high-order address lines. Writing a byte to address F000h in RAM will raise and lower A15 with no damage to RAM contents in low memory. The LS175 dual flip-flops (IC A) aligns the clock information with the data from the keyboard.

We are almost there, hardware-wise! All that remains to be done is to have the Z80 process the data and pass it on to the main computer. On my system, the keyboard input is from a SD Systems 8024 Video board. This board requires a keyboard with a parallel port. The data must be strobed into this port by a positive-going pulse. I have modified this board slightly so that once data is read from this port, a negative-going pulse is sent back to the smart keyboard interface letting it know the data has been read. Other boards may have different strobe protocols. These can often simply be accommodated by one or two 74LS04 inverters in the circuit. For those computer systems that have a serial keyboard, you could replace one of the PIO's with a Zilog SIO. This is a simple hardware replacement, but you should carefully study the software setup procedure to talk to the SIO.

Pin 26 of the Z80 (reset) is connected to the main-computer reset line. In this way, a reset to the main computer also resets the smart keyboard circuit. Any keyboard characters in a queue or taken through a translation table are flushed out in this process. The NMI* pin of the Z80 is connected to a one-bit output port of the main computer. The function of this will become clear when we discuss the software used to drive this board. Other pins of the Z80 are left either unconnected or tied high via a 1K resistor.

SOFTWARE

Writing software for a computer like this is a lot of fun. Because you have complete control of the smart-keyboard interface Z80 at all times, you can place values in certain registers and know they are always going to be there. We can, for example, really make use of the Z80 alternate registers. The software I used to program this board has been submitted to the public-domain SIG/M users group and will be available in one of their future releases. What I would like to do, is step you through the main points. The complete details for all routines can be found in the public-domain code itself. The first two lines of code start off:

```
ORG     00H
LD      SP,STACK
```

Since the Z80 reads an opcode from memory location 0, at power on or a reset, the ROM code must originate here. First we need a valid location for the stack. To be on the safe side, we will put the stack high up in RAM, but just below certain reserved RAM memory locations. I have used STACK = 0FF0h.

Next we initialize the Zilog PIO's. The 2 PIO's have 4 ports which can be configured in software as input, output, or bidirectional. Consult the Zilog technical literature for a detailed explanation of what this entails. We will set up both PIO port A's as output ports (MODE 0) and port B's as input ports (MODE1).

The PIO's also have the capability of generating an interrupt under certain data-transfer conditions. We do not need this here and so we must program the chips to disable this function.

Programming the chip is easy. You select the appropriate PIO control port (data port + 1 in this example) and send two bytes of code. Since we have 4 ports in all, we must send 2 bytes to each of 4 ports.

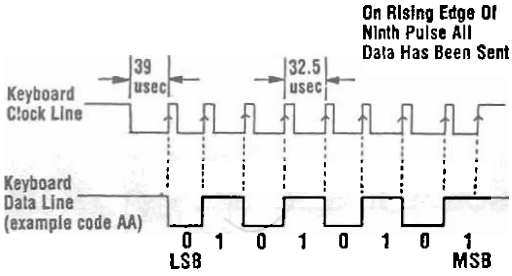


Figure 2. Timing diagram of serial data sent from a Keytronics IBM-compatible keyboard.

3B BB	3C BC	01 81	02 82	03 83	04 84	05 85	06 86	07 87	08 88	09 89	0A 8A	0B 8B	0C 8C	0D 8D	0E 8E	45 C5		46 C6					
3D BD	3E BE	0F 8F	10 90	11 91	12 92	13 93	14 94	15 95	16 96	17 97	18 98	19 99	1A 9A	1B 9B	29 A9	47 C7	48 C8	49 C9	4A CA				
3F BF	40 C0	1D 9D	1E 9E	1F 9F	20 A0	21 A1	22 A2	23 A3	24 A4	25 A5	26 A6	27 A7	28 A8	1C 9C		4B CB	4C CC	4D CD	4E CE				
41 C1	42 C2	2B AB	2A AA	2C AC	2D AD	2E AE	2F AF	30 B0	31 B1	32 B2	33 B3	34 B4	35 B5	36 B6	37 B7	4F CF	50 D0	51 D1	1C 9C				
43 C3	44 C4	38 B8	39 B9											3A BA		52 D2		53 D3					

Figure 3. Keyboard code chart for a Keytronics IBM-compatible keyboard. The upper number on each key is for downstrokes and lower number is for key upstrokes.

This is done as shown in Listing 1.

Having set-up the PIO's, it is a good practice to clear them of any false data they may have acquired before or during initialization. We do this by:

```
IN      a,(DATA$B)
IN      a,(DATA$C)
```

Next we have to enable the Z80 in the correct interrupt mode. Again you should consult the Zilog literature if you do not understand how this is done. In our case we need interrupt mode 1. This will cause the CPU to jump to location 38H in our ROM anytime the INT* (pin 16) on the Z80 is pulled low. At that location will be the code that will get data from the PIO port and process it. An INT will occur only when the hardware has received 8 serial bits of data from the keyboard (the low at pins 1 & 2 of the 74LS174-B have been shifted 8 times). The code at 38H is shown in Listing 2.

What we are doing here, is quickly taking the data byte at the keyboard and placing it in a cyclic 256-byte buffer in RAM memory. We do not know when such an INT would occur and so cannot count on what is in the "main" Z80 registers. For this purpose, we set aside the alternative Z80 registers. As described below, the keyboard data port will ALWAYS be in register C' and HL' will ALWAYS point to the end of the incoming character queue. One nice thing about the INC L, is that it insures the queue will always wrap around after 256 bytes. We do not have to move pointers back to the beginning of the queue once they reach its end. We have already discussed the use of address line A15 to reset the 74LS164's. We waste the IY reg to do just this in this application. It is setup with the value F000h. As you can see from the timing diagram of the clock and data lines in Figure 2, there is one extra clock pulse we have to absorb, before the keyboard is finished sending its byte of data. This is monitored at bit 1 of the PIO 2, port-B bit 1 (Figure 3).

At org 100H in the ROM, we have the code (Listing 3) to set-up the computer and keep it happy while it is waiting for a character.

What we have done in the above code is set-up the register pairs HL, DE, BC, and HL' to point to two regions in RAM memory that will contain the incoming and outgoing keyboard data.

Here is what will happen. When an INT occurs, a keyboard character will be placed at the end of a queue in the inbuffer. The pointer to this queue (in HL') will be increased by one byte and a 0 placed in that memory location. When the Z80 is not getting more characters from the keyboard and placing them in the inbuffer or sending them to the main computer by reading from the outbuffer (see below), it is checking if its pointer to the character in the inbuffer is zero. If it is not, it assumes one or more new characters have arrived. These characters are read from the inbuffer, translated if need be (see below), and placed in the outbuffer. The pointers to both buffers are updated accordingly and a zero flag in each buffer is set to indicate the ends of the buffers. It is important to remember that this is all this Z80 will ever have to do. So, certain registers can be set aside permanently to hold certain values. The code is shown in Listing 4.

The subroutine TRANS is the heart of the code. It takes the bit pattern from the keyboard and translates it into ASCII characters. This is necessary because the IBM/Keytronics keyboards sends only a binary number representation of the key pressed, not the ASCII character. For example, the ESC key sends 01H, the "1" key 02H, etc. Further, the keyboard distinguishes between key down-strokes and key up-strokes. Up-strokes have the same binary number plus 80H. In other words, their most significant bit is set. This is not all as bad as it sounds. It means we must make a lookup table of ASCII characters from binary values. Figure 3 shows the way the keys are numbered on an IBM/Keytronics keyboard. Part of the corresponding table looks like:

```
IBMTBL: DB      0
        DB      1BH,'1234567890-=',8H
        DB      9H,'qwertyuiop[]',0DH
        DB      42H,'asdfghjkl;',27H,60H
        DB      41H,'\zxcvbnm,./',41H,'*'
        DB      1EH,' ',44H
```

When TRANS arrives, with say a 02H in register A, the actual ASCII value is obtained by adding 2 to the [HL] register pair which is pointing to the start of the table. Then an instruction:

```
LD      A,(HL)
```

places the correct ASCII character in register A. The code for TRANS is as shown in Listing 5.

For the Smart Keyboard to be of use, we need a number of tables of the type described above. This is because the meaning of a key-board character can change, depending on whether keys such as the shift, lock, NUM or control keys were previously pressed. Each time one of these keys is pressed the appropriate flag is updated in RAM to set our [HL] pointer to the appropriate table. Rather than present all this code, I have sent it to the public domain SIG/M library. The file SKEY.Z80 contains all the code described in this article.

Besides the above special keys, which almost every keyboard has, we can add new ones. For example, we might have a case in which the F1 key is pressed; we have TRANS point to a table which defines the keys of the number keypad as special control sequences for a word processor such as Wordstar. F2 would point to a different table for a text editor such as Vedit. For complete compatability with the IBM-PC, we can have a table where untranslated information (binary key numbers) is sent to the BIOS of the computer and translated exactly as IBM describes.

Now for the most important feature of the board — single-key to multi-key translation. If TRANS observes that the translated key is greater in value than 7FH (bit 7 high), another routine which I have named MULTI, is called. This routine looks at the "special character" and depending on its value, places not one, but a string of characters in the outbuffer. This string is then read by the main computer which thinks they were individually typed. To give you an example: If I press the "F6 key" TEST.TXT on my keyboard - 9 key strokes +CR, the computer would receive VEDIT TEST.TXT. At the same time, the numeric keypad would be automatically configured for the Vedit cursor/editing control sequences. This is done by pointing TRANS to the appropriate lookup table.

Because we have the power of a microprocessor at our disposal, we can do a lot with one keystroke. For example, I like to have the same cursor control keys for all my editors and word processors. Vedit uses one control character in many situations in which Wordstar uses two. In fact, in some cases, one needs to toggle two sets of dual control characters in Wordstar to get the same effect as with Vedit. This can be easily accommodated with this setup. The routine MULTI is quite long and contains a number of special-case treatments for special keys. There is not room here to present the whole routine. However the kernel of the routine is shown below in Listing 6.

In this routine, two data areas are used. Multi$table contains a list of pointers to the first character of each string. The offset into multi$table X2 will allow the correct pointer to be picked up. This in turn points to the actual string which can be of varied length. A simplified version of the table is shown in Listing 7.

The DROP routine then transfers each character of the string into the outbuffer until a 0 is reached.

One final point. When you are finished with your special application program, it would be nice to reset the keyboard back to its default configuration (CP/M or MS-DOS). It would be nice also not to have to do this manually. This is where I have utilized the NMI line to the Z80. Whenever this line is pulled low, the Z80 stops whatever it is doing and jumps to 66H in RAM where it finds the code shown in Listing 8.

Now, while this is not entirely clear unless you read the complete code, suffice to say, all flags set up in memory are reset to their initial power-on (CP/M or MS-DOS) configuration. How does the NMI line get triggered? This is where you have to use your own initiative. In my S-100 system, I use one bit of an output port to lower and then raise the NMI line on the Z80 board. In the BIOS portion of CP/M+ and CP/M86, one can put the required code in the warm-start module just before control is transferred to the CCP.

For MS-DOS, I utilize the fact that the command-line prompt is definable. For example it could be "A>" or "A$". I set it to "Esc Esc A>". When my console output driver sees the unique "Esc Esc" sequence, it sends a signal to the smart keyboard to pulse the NMI line of the Z80. There are many other possibilities. Many editors have logoff control sequences, indeed, you could set aside one keyboard key for the function.

Lastly, the power of the Z80 in this application is hardly used. You can easily modify the design to use keys to switch-on drive motors, CRT's, or to restrict access to certain programs. I have mine connected to a speech synthesizer that sends me all kinds of information and reminders. (μ)

LISTING 1

```
            LD      HL,PIOTBL       ;POINT TO THE PIO TABLE
OUTBLOCK:   LD      A,(HL)          ;GET BYTE COUNTER
            OR      A               ;TEST FOR END OF TABLE
            JR      Z,ALLSET        ;0 WHEN ALL PORTS DONE
            LD      B,A             ;ELSE PUT COUNT IN B
            INC     HL              ;POINT TO PORT #
            LD      C,(HL)          ;PUT PORT IN [C]
            INC     HL              ;POINT TO FIRST DATA VALUE
            OTIR                    ;SEND 2 BYTES TO PORT IN [C]
            JR      OUTBLOCK        ;GO TO NEXT PORT.
```

Continued on next page.

John Monahan is a molecular biologist. He received a PhD in Bio-Chemistry from MacMaster University in Ontario Canada. John has been a computer hobbyist for the past ten years and has built several homebrew systems. He has been a member of the Amateur Computer Group of New Jersey for over 9 years and recently moved to the San Francisco bay area. He can be contacted at: Box 1908, Orinda CA 94563.

```
EXX                     ;SETUP THE REGS FOR INT PROCESSING
LD      HL,INBUFFER     ;[HL'] POINTS TO BUFFER
LD      C,DATA$B        ;DATA PORT FOR KEYBOARD
LD      (IY+0),A        ;CLEAR THE 74LS164'S
EXX
EI                      ;CAN NOW ENABLE INTS.

PIOTBL:-
        DB      2       ;Number of bytes to send
        DB      PORT$A  ;value of port A
        DB      0FH     ;mode 0
        DB      03H     ;No Interrupts.
        DB      2       ;Number of bytes to send
        DB      PORT$B  ;value of port B
        DB      4FH     ;mode 1
        DB      03H     ;No Interrupts.
        DB      2       ;Number of bytes to send
        DB      PORT$C  ;value of port C
        DB      0FH     ;mode 0
        DB      03H     ;No Interrupts.
        DB      2       ;Number of bytes to send
        DB      PORT$A  ;value of port D
        DB      0FH     ;mode 0
        DB      03H     ;No Interrupts.
        DB      0       ;end of table flag
```

LISTING 2

```
CLOCK:  EXX             ;SAVE ALT REGS
        EX      AF,AF'
        IN      A,(C)   ;DEPOSIT CHARACTER AT END OF QUEUE
        LD      (HL),A
        INC     L       ;POINT TO NEXT BYTE OF 256 BYTE QUEU
        XOR     A       ;FLAG END OF BUFFER
        LD      (HL),A
CLOCK1: IN      A,(DATA$D) ;WAIT FOR LAST CLOCK PULSE TO RISE
        BIT     1,A
        JR      NZ,CLOCK1
CLOCK2: IN      A,(DATA$D)
        BIT     1,A
        JR      Z,CLOCK2  ;RAISE A15 LINE TO RESET 74LS164'S
        LD      (IY+0),A  ;RESTORE ALT REGS
        EX      AF,AF'
        EXX             ;ENABLE ANY MORE INTS
        EI              ;RETURN FROM INTS
        RETI
```

LISTING 3

```
BEGIN:  XOR     A
        LD      (SHIFTFLAG),A   ;FLAG TO INDICATE KEYS IN SHIFT MODE
        LD      IY,0F000H       ;FOR A15 TO CLEAR 74LS164'S
        LD      HL,INBUFFER     ;BUFFER FOR INCOMING CHARACTERS
BEGNMI: LD      (HL),A          ;ZERO AT END OF BUFFER
        LD      (WORD1),A       ;SPECIAL KEY FLAGS USED BELOW
        LD      (WORD2),A
        LD      (FLAG1),A
        LD      (FLAG2),A
        LD      DE,OUTBUFFER    ;END OF OUTBUFFER QUEUE
        LD      BC,OUTBUFFER    ;START OF OUTBUFFER QUEUE
        LD      (DE),A          ;ZERO IT ALSO
```

```
LOOP:   LD      A,(HL)          ;SEE IF ANYTHING IN INBUFFER
        OR      A
        JR      Z,LOOP1         ;IF NOTHING CHECK OUTBUFFER
        INC     L               ;INCREASE POINTER FOR NEXT TIME
        PUSH    HL              ;SAVE THESE REGISTERS
        PUSH    BC
        CALL    TRANS           ;CONVERT BIT PATTERN TO ASCII CHARS
        POP     BC              ;RESTORE REGISTERS
        POP     HL
        OR      A               ;DO NOT SEND NULLS TO MAIN COMPUTER
        JR      Z,LOOP1
        BIT     7,A             ;FLAG TO INDICATE SPECIAL CASES
        JR      Z,NORMAL        ;IF Z THEN SIMPLE ONE TO ONE
        PUSH    HL              ;NEED TO SAVE IT AGAIN
        CALL    MULTI           ;ONE IN CHAR TO SEVERAL OUT CHARS
        POP     HL
        JR      LOOPM

;
NORMAL: LD      (DE),A          ;PUT ASCII CHAR IN OUTBUFFER
        INC     E               ;FLAG END WITH ZERO
LOOPM:  XOR     A
        LD      (DE),A

;
LOOP1:  LD      A,(BC)          ;ANYTHING TO SEND TO
        OR      A

        JR      Z,LOOP          ;NOTHING THERE BACK TO INBUFFER QUEUE
;
LOOP2:  IN      A,(DATA$D)      ;IS COMPUTER READY FOR NEXT CHAR.
        BIT     0,A
        JR      NZ,LOOP2        ;HANG IN THERE UNTIL READY
        LD      A,(BC)          ;[BC] POINTS TO CHAR
        INC     C               ;FOR NEXT TIME
        OUT     (DATA$A),A      ;SEND IT
        JR      LOOP
```

LISTING 5

```
;
TRANS:  LD      HL,SHIFTFLAG    ;POINT TO  FLAG OF CURRENT SHIFT ETC. MODES
        BIT     7,A             ;IS KEY AN UPSTROKE MOVE
        JR      NZ,UPSTROKE
        CP      SHIFT1          ;IS IT A SHIFT KEY
        JR      Z,ISSHIFT
        CP      SHIFT2          ;TWO SHIFT KEYS ON BOARD
        JR      Z,ISSHIFT
        CP      SHIFT3          ;FOR SHIFT LOCK KEY
        JP      Z,ISLOCK
        CP      NUMLOCK         ;IS IT NUMBER LOCK KEY
        JP      Z,ISNUM
        CP      CTRL            ;IS IT THE CTRL KEY
        JP      Z,ISCTRL
        PUSH    AF              ;SAVE CHARACTER FOR THE MOMENT

        LD      A,(HL)          ;FIND OUT WHICH TABLE IS CURRENT
        BIT     6,A             ;BIT 6=1 FOR WORDSTAR TABLE
        JR      Z,NOTSTAR
        LD      HL,STARTABLE
        JR      ENDSHIFT
NOTSTAR:BIT     4,A             ;BIT 4=1 FOR VEDIT TABLE
        JR      Z,NOTVED
        LD      HL,VEDTABLE
        JR      ENDSHIFT
NOTVED: BIT     5,A             ;BIT 5=1 FOR IBM/DOS KEYBOARD
        JR      Z,NOTIBM
        LD      HL,IBMTABLE
        JR      ENDSHIFT
NOTIBM: LD      HL,CPMTABLE     ;X000XXXX = DEFAULT TABLE TO CP/M TABLE

;
ENDSHIFT:AND    00000010B       ;IS UPPER OR LOWER SECTION OF TABLE REQ
        JR      Z,UPPERHALF     ;BIT 1=1 FOR UPPER CASE CHARS (!@#$...)
        LD      B,0
        LD      C,HALF
        ADD     HL,BC           ;HL NOW POINTS TO SECOND HALF OF EACH TABLE
UPPERHALF:
        LD      B,0
        POP     AF              ;GET CHARACTER BIT PATTERN
        LD      C,A
        HL,BC                   ;GET OFFSET INTO TABLE
        C,(HL)                  ;STORE CHARACTER BACK IN [C]
        HL,SHIFTFLAG            ;NOW NEED TO SEE IF WE NEED SHIFT MODE
        A,(HL)
        AND     00000110B       ;SEE IF a....z TO UC IS REQ
        JR      Z,ENDSHF        ;BITS 1 OR 2 =1 FOR UPPER CASE
        LD      A,C
        CP      'a'
        JR      C,ENDSHF
        CP      'z'+1
        JR      NC,ENDSHF
        SUB     20H
        LD      C,A
ENDSHF: BIT     3,(HL)          ;ARE CONTROL CHARACTERS REQ.
        LD      A,C             ;BIT 3=1 FOR CTRL CHARS.
        RET     Z
        AND     00011111B
        RET

;
UPSTROKE:CP     UPSH1           ;WERE ANY SPECIAL KEYS RELEASED
        JR      Z,NOTSHIFT
        CP      UPSH2
        JR      Z,NOTSHIFT
        CP      UPCTRL
        JR      Z,NOTCTRL
        XOR     A               ;NONE THEN NORMAL RELAESE
        RET

;
ISSHIFT:SET     1,(HL)          ;IS SHIFT KEY SET FLAG
        XOR     A
        RET

;
ISLOCK: XOR     A               ;TOGGLE CAPS LOCK ON/OFF
        BIT     2,(HL)
        JR      Z,TOGGLE
        RES     2,(HL)
        RET
TOGGLE: SET     2,(HL)
        RET
;
ISNUM:  BIT     0,(HL)          ;THIS HAS TOGGLE INFO FOR NUMLOCK LED/BIT
        JR      NZ,TOGG1
        LD      A,(HL)          ;IF 0, FORCE IBM TABLE
        AND     00001110B
        or      00100001B       ;SO NEXT TIME CP/M TABLE
        LD      (HL),A
        call    cleanflags      ;CLEAR VEDIT & WS FLAGS
        LD      HL,SPIBM
        CALL    TALK
        XOR     A
        RET
TOGG1:  LD      A,(HL)          ;IF 1, FORCE CP/M TABLE
        AND     00001110B       ;NOTE BIT 0 ALSO SET TO Z
        LD      (HL),A
        call    cleanflags
        LD      HL,SPCPM
        CALL    TALK
        XOR     A
        RET
```

```
                                        LISTING 6
ISCTRL: XOR     A               ;IS A CONTROL KEY
SET FLAG
        SET     3,(HL)
        RET
;
NOTCTRL:XOR     A               ;TURN OFF CONTROL KEY FLAG
        RES     3,(HL)
        RET
;
NOTSHIFT:XOR    A               ;TURN OFF SHIFT FLAG
        RES     1,(HL)
        RET
                                        LISTING 7
MULTI:                          ;TREAT SPECIAL CASE SITUATIONS
;.....
;   code for special case situations.
;.....
;                               ;NOW TREAT "NORMAL" SINGLE KEY TO MULTI
;                               ;CHARACTER STRINGS TRANSLATIONS.
mul3:   and     7fh             ;STRIP OFF HIGH (FLAG) BIT
multil: ld      hl,multi$table  ;POINT TO LIST OF STRINGS
        rlca                    ;X2 FOR OFFSETS
        push    bc
        ld      c,a             ;DO 16-BIT ADD
        ld      b,0
        add     hl,bc           ;[HL] NOW POINTS TO STRING ADDRESS
        ld      c,(hl)          ;TEMPORARY STORE FOR L
        inc     hl
        ld      h,(hl)          ;[HL] NOW POINTS TO STRING ITSELF
        ld      l,c
        pop     bc
;
DROP:   LD      A,(HL)          ;PICK UP STRING CHARACTERS
        OR      A               ;END OF STRING YET?
        RET     Z
        LD      (DE),A          ;DROP IN STRING CHARACTERS IN OUTBUFFER
        INC     E               ;REMEMBER CYCLIC LOOP 0...255
        INC     HL              ;NEXT CHAR IN STRING
        JR      DROP
                                        LISTING 7
multi$table:
        DW      VEDIT$                  ;80H FOR VEDIT STRING
        DW      MSDOS$                  ;81H FOR MSDOS STRING
        DW      CHDIR$                  ;82H FOR CHDIR STRING
        DW      SETDEF$                 ;83H FOR SETDEF STRING
        .....
        DW      DIRA$                   ;A8H FOR DIR STRING
;
VEDIT:  DB      'VEDIT ',0              ;SAMPLE STRINGS
MSDOS:  DB      'MSDOS ',0
CHDIR:  DB      'CHDIR ',0
SETDEF: DB      'SETDEF *,A:[ORDER=(CMD,SUB),DISPLAY]',0
        .....
DIRA:   DB      'DIR A: [FULL]',0
                                        LISTING 8
                ORG     66H
NMILOC: LD      SP,STACK
        LA      A,(SHIFTFLAG)   ;GET SHIFT/TABLE FLAGS
        AND     00001111B       ;RESET ALL FLAGS
        LD      (SHIFTFLAG),A
        XOR     A
        JP      BEGNMI
```