```
MOV AL,0FDH
OUT 21H,AL
```

Keep in mind that the state of the *interrupt flag* within the 8088 will ultimately determine whether or not any interrupt signal is received.

The second 8259 programming action that we must be concerned with is the signaling of the end of an interrupt service routine. This is accomplished by sending the "end of interrupt" (EOI) command, represented by 20H, to the interrupt command register within the 8259. Coincidentally, this one-byte register is accessed via i/o port 20H. That is all there is to controlling the interrupt mechanism! A complete example will appear later in this chapter.

THE 8255 PROGRAMMABLE PERIPHERAL INTERFACE

The 8255 is a general-purpose i/o interface chip that can be configured in many different ways. It is used on the system board to support a variety of devices and signals. These include the keyboard, speaker, configuration switches, and several other signals.

The chip contains three ports, called PA, PB, and PC. These are mapped to i/o addresses 60H, 61H, and 62H, respectively. In addition, there is a one-byte command register on the chip, accessed via i/o address 63H. On power-up, the BIOS initializes this chip by sending a value of 99H to the command register. This configures the 8255 so that PA and PC are considered input ports and PB is considered an output port. The meaning of each port is defined in Fig. 5-4. Note that additional logic on the system board allows us to select alternate inputs to ports PA and PC by setting certain bits in output port PB. In addition, we can read back the last value that was written to port PB by performing an input operation on port PB.

Fig. 5-5 gives an example of how we might make use of this hardware to read the settings of the configuration switches. There are two configuration switches on the system board; each can be set manually to represent any one-byte value. They are normally set up to indicate the various hardware options installed in the Personal Computer system. If, for example, our program needed to know how many disk drives were attached to the system, it could examine the two high-order bits of switch 1. This is accomplished by the program instructions of Fig. 5-5. Note that to enable the configuration-switch information onto port PA, we must first set bit 7 of port PB.

THE KEYBOARD

The system board provides an interface to the Personal Computer keyboard via the interrupt mechanism and ports PA and PB of the 8255 chip. This hardware is normally supported and controlled by programs running in the BIOS so that we do not have to be concerned with it. We simply

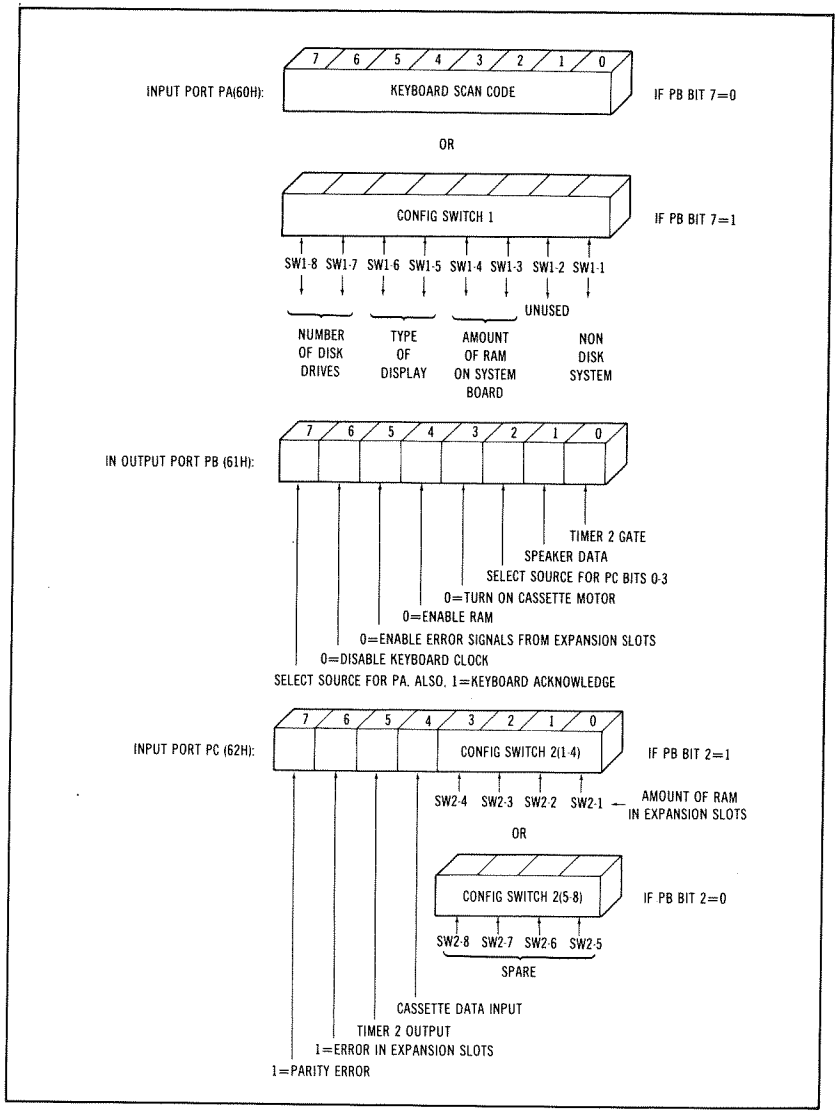

Fig. 5-4. 8255 port allocations.

access the keyboard via BIOS INT 16H, as shown in the last chapter. By understanding the hardware, however, we can write our own keyboard-support software, with certain interesting advantages.

```
        IN      AL,61H      ;GET PRESENT VALUE OF PORT PB
        OR      AL,80H      ;FORCE BIT 7 ON
        OUT     61H,AL      ;SET PORT PB BIT 7 = 1 ;
;                           CONFIG SW1 NOW GATED TO PORT PA
        IN      AL,60H      ;READ PORT PA = CONFIG SWITCH 1
        NOT     AL          ;INVERT BITS
        MOV     CL,6        ;SET UP SHIFT AMOUNT
        SHR     AL,CL       ;ISOLATE BITS 7,6 OF AL
; NOW AL = NUMBER OF DISK DRIVES ATTACHED TO SYSTEM,
; AS OBTAINED FROM CONFIG SWITCH 1, POSITIONS 8,7.
```

Fig. 5-5. Reading the configuration switches.

Within the keyboard itself is a small microprocessor that scans for and detects any change in state of the keys. This processor receives its basic power and clock signals from the system board. We can disable the clock signal going to the keyboard by setting bit 6 of port PB to 0. This will prevent the keyboard from operating. In addition, we can send an *acknowledge signal* to the keyboard by setting bit 7 of port PB to 1. To ensure that the keyboard is properly enabled, we must set bits 7 and 6 of port PB to 0 and 1, respectively. In this state, the keyboard will generate an interrupt signal (IRQ1) whenever any key is depressed or released. It will then transmit a one-byte *scan code* to the system board and wait for the acknowledge signal to be returned. The scan code will be a number between 1 and 83 that uniquely identifies which key changed state (there are 83 keys on the keyboard). The high-order bit (bit 7) of the scan code indicates whether the key was depressed or released. It will be 0 if the key was depressed, and it will be 1 if the key was released. Fig. 5-6 identifies the scan code associated with each key on the keyboard.

It is the responsibility of the keyboard-support software to detect the keyboard interrupt and to respond to it as follows. First, the scan-code value transmitted to the system board must be obtained by reading from 8255 port PA. Then, the acknowledge signal must be sent back to the keyboard by momentarily setting bit 7 of port PB. The scan code itself may be interpreted in any manner desired. Thus, the meaning of each key can be defined, or even dynamically changed, by software. A more important consideration, however, stems from the fact that the keyboard interrupt occurs *asynchronously* with respect to the main program running in the computer. What this means is that the striking of a key (and its subsequent handling by the keyboard-support software) can occur at any time, and it is totally independent of when the main program may wish to receive keyboard input. Our keyboard support routine is therefore required to *buffer*, or save, any keyboard input that it receives. To accomplish this, we employ a "first-in, first-out" buffer, also referred to as a *circular queue*.

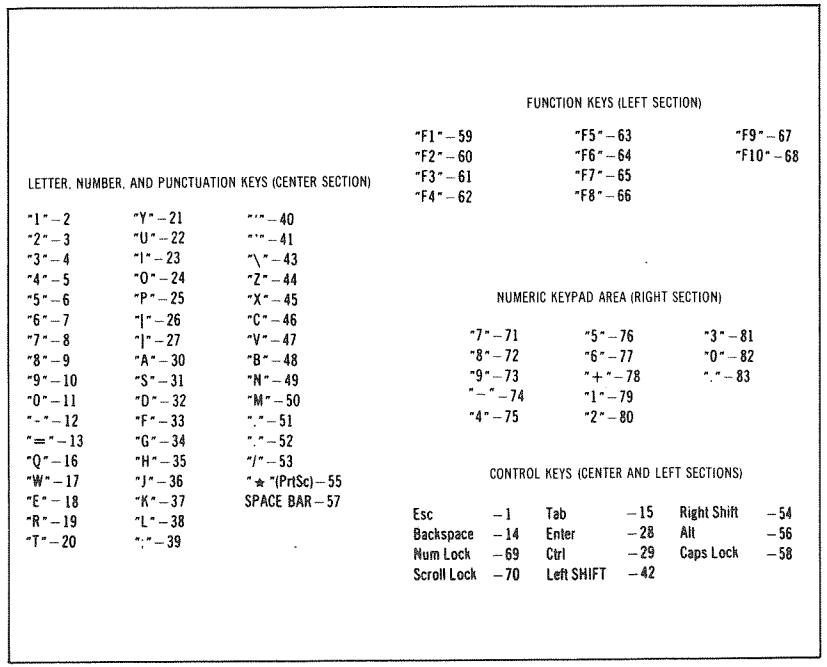

Fig. 5-6. Keyboard scan codes (listed in decimal).

Scan codes received from the keyboard are converted into the appropriate ASCII character codes and then placed onto this queue. When the main program wishes to obtain keyboard input, it calls an auxiliary routine within the keyboard-support software. This routine takes the characters off the queue, *in the order in which they were received,* and passes them to the main program. The size of the queue determines the maximum number of characters that can be buffered at any time. This represents the number of keystrokes that you can "type ahead" of the main program.

In Fig. 5-7, a complete program that sets up and utilizes its own keyboard-support software is presented. The program is kept relatively simple by omitting features normally handled by BIOS keyboard support, such as upper/lower-case alphabetics, "shift" and "shift-lock" keys, and special control-key combinations. The main program consists of two parts. Part one modifies the interrupt-service–routine address table to point to our own keyboard interrupt routine. It is also responsible for initializing the necessary hardware interfaces by sending commands to the 8259 and 8255 chips. Once this has been accomplished, we enter part two, a simple loop that reads keyboard input and displays it on the screen. The other

```
00010 ;
00020 ;
00030 ; EXAMPLE OF CUSTOM KEYBOARD SUPPORT SOFTWARE
00040 ;
00050 STACK     SEGMENT PARA STACK 'STACK'
00060   DB      256 DUP (0)     ;256 BYTES OF STACK SPACE
00070 STACK     ENDS
00080 ;
00090 DATA      SEGMENT PARA PUBLIC 'DATA'
00100 BUFFER    DB      10 DUP (0)      ;TEN BYTE KEYBOARD BUFFER
00110 BUFPTR1 DW        0               ;POINTS TO START OF BUFFER
00120 BUFPTR2 DW        0               ;POINTS TO END OF BUFFER
00130 ; NOTE: WHEN BUFPTR1 = BUFPTR2 , THEN THE BUFFER IS EMPTY.
00140 ; SCANTABLE CONVERTS SCAN CODES RECEIVED FROM THE KEYBOARD
00150 ;  INTO THEIR CORRESPONDING ASCII CHARACTER CODES:
00160 SCANTABLE DB      0,0,'1234567890-=',8,0
00170   DB      'QWERTYUIOP[]',0DH,0
00180   DB      'ASDFGHJKL;',0,0,0,0
00190   DB      'ZXCVBNM,./',0,0,0
00200   DB      ' ',0,0,0,0,0,0,0,0,0,0,0,0,0
00210   DB      '789-456+1230.'
00220 DATA      ENDS
00230 ;             '
00240 CODE      SEGMENT PARA PUBLIC 'CODE'
00250 START     PROC    FAR
00260 ;
00270 ; STANDARD PROGRAM PROLOGUE
00280 ;
00290   ASSUME  CS:CODE
00300   PUSH    DS      ;SAVE PSP SEG ADDR
00310   MOV     AX,0
00320   PUSH    AX      ;SAVE RET ADDR OFFSET (PSP+0)
00330   MOV     AX,DATA
00340   MOV     DS,AX   ;ESTABLISH DATA SEG ADDRESSABILITY
00350   ASSUME  DS:DATA
00360 ;
00370 ; PART1: SETUP OUR OWN KEYBOARD INTERRUPT SERVICE ROUTINE
00380 ;
00390   CLI             ;DISABLE ALL INTERRUPTS
00400   MOV     AX,0
00410   MOV     ES,AX   ;POINT EXTRA SEGMENT AT THE...
00420 ;         ...INTERRUPT SERVICE ROUTINE ADDRESS TABLE
00430   MOV     DI,24H  ;OFFSET OF ENTRY FOR TYPE CODE 09H
00440   MOV     AX,OFFSET KBINT ;OFFSET OF OUR SERVICE ROUTINE
00450   CLD                     ;SET 'FORWARD' STRING OPERATIONS
00460   STOSW                   ;PLACE IT IN THE TABLE
00470   MOV     AX,CS           ;SEG OF OUR SERVICE ROUTINE
00480   STOSW                   ;PLACE IT IN THE TABLE
00490   MOV     AL,0FCH ;ENABLE TIMER AND KYBD IRUPTS
00500   OUT     21H,AL  ;WRITE INTERRUPT MASK REGISTER
00510   STI             ;ENABLE INTERRUPTS TO THE 8088
00520 ;
00530 ; PART2: READ FROM KEYBOARD AND DISPLAY CHARS ON SCREEN
00540 ;
```

Fig. 5-7. Custom keyboard-support program.

Continued on next page.


```
00550 FOREVER: CALL     KBGET   ;WAIT FOR A CHARACTER FROM THE KEYBOARD
00560   PUSH    AX      ;SAVE THE CHARACTER
00570   CALL    DISPCHAR ;DISPLAY THE CHARACTER RECEIVED
00580   POP     AX      ;RESTORE THE CHARACTER
00590   CMP     AL,0DH  ;WAS IT A CARRIAGE RETURN?
00600   JNZ     FOREVER ;BRANCH IF NOT
00610   MOV     AL,0AH  ;YES IT WAS, WE MUST ALSO DISPLAY...
00620   CALL    DISPCHAR ;...A LINE FEED!
00630   JMP     FOREVER ;STAY IN THIS LOOP FOREVER
00640 ;
00650 ; CALL KBGET TO WAIT FOR A CHARACTER TO BE RECEIVED FROM
00660 ;  THE KEYBOARD.  THE CHARACTER IS RETURNED IN REG AL.
00670 KBGET     PROC    NEAR
00680   PUSH    BX              ;SAVE REGISTER BX
00690   CLI                     ;DISABLE INTERRUPTS
00700   MOV     BX,BUFPTR1      ;START OF BUFFER
00710   CMP     BX,BUFPTR2      ;IS BUFFER EMPTY?
00720   JNZ     KBGET2          ;->NO
00730   STI                     ;RE-ENABLE INTERRUPTS
00740   POP     BX              ;RESTORE REGISTER BX
00750   JMP     KBGET           ;WAIT UNTIL SOMETHING IN BUFFER
00760 ; THERE IS SOMETHING IN THE BUFFER, GET IT :
00770 KBGET2: MOV       AL,[BUFFER+BX]  ;GET CHAR AT BUFFER START
00780   INC     BX              ;INCREMENT BUFFER START
00790   CMP     BX,10           ;HAVE WE WRAPPED AROUND?
00800   JC      KBGET3          ;BRANCH IF NOT
00810   MOV     BX,0            ;YES, WRAP AROUND
00820 KBGET3: MOV       BUFPTR1,BX      ;INDICATE NEW START OF BUFFER
00830   STI                     ;RE-ENABLE INTERRUPTS
00840   POP     BX              ;RESTORE REGISTER BX
00850   RET                     ;RETURN FROM KBGET
00860 KBGET     ENDP
00870 ;
00880 ; KBINT IS OUR OWN KEYBOARD INTERRUPT SERVICE ROUTINE:
00890 ;
00900 KBINT     PROC    FAR
00910   PUSH    DS      ;SAVE ALL ALTERED REGISTERS!!
00920   PUSH    BX
00930   PUSH    AX
00940 ;
00950 ; ESTABLISH ADDRESSABILITY TO OUR DATA SEGMENT:
00960 ;
00970   MOV     AX,DATA
00980   MOV     DS,AX
00990 ;
01000 ; READ THE KEYBOARD DATA AND SEND THE ACKNOWLEDGE SIGNAL:
01010 ;
01020   IN      AL,60H  ;READ KEYBOARD INPUT
01030   PUSH    AX      ;SAVE KEYBOARD INPUT
01040   IN      AL,61H  ;READ 8255 PORT PB
01050   OR      AL,80H  ;SET KEYBOARD ACKNOWLEDGE SIGNAL
01060   OUT     61H,AL  ;SEND KEYBOARD ACKNOWLEDGE SIGNAL
01070   AND     AL,7FH  ;RESET KEYBOARD ACKNOWLEDGE SIGNAL
01080   OUT     61H,AL  ;RESTORE ORIGINAL 8255 PORT PB
```

Fig. 5-7 (cont). Custom keyboard-support program.

Continued on next page.

```
01090 ;
01100 ; DECODE THE SCAN CODE RECEIVED:
01110 ;
01120   POP     AX      ;REGAIN THE KEYBOARD INPUT (AL)
01130   TEST    AL,80H  ;IS IT A KEY BEING RELEASED?
01140   JNZ     KBINT2  ;BRANCH IF YES, WE IGNORE THESE
01150   MOV     BX,OFFSET SCANTABLE ;SCAN CODE - ASCII TABLE
01160   XLATB           ;CONVERT THE SCAN CODE TO AN ASCII CHAR
01170   CMP     AL,0    ;IS IT A VALID ASCII KEY?
01180   JZ      KBINT2  ;BRANCH IF NOT
01190 ;
01200 ; PLACE THE ASCII CHARACTER INTO THE BUFFER:
01210 ;
01220   MOV     BX,BUFPTR2      ;GET POINTER TO END OF BUFFER
01230   MOV     [BUFFER+BX],AL  ;PLACE CHAR IN BUFFER AT END
01240   INC     BX              ;INCREMENT BUFFER END
01250   CMP     BX,10           ;HAVE WE WRAPPED AROUND?
01260   JC      KBINT3          ;BRANCH IF NOT
01270   MOV     BX,0            ;YES, WRAP AROUND
01280 KBINT3: CMP       BX,BUFPTR1      ;IS BUFFER FULL?
01290   JZ      KBINT2          ;BRANCH IF YES, WE LOSE THIS CHAR
01300   MOV     BUFPTR2,BX      ;INDICATE NEW END OF BUFFER
01310 ;
01320 ; NOW INDICATE "END OF INTERRUPT" TO THE INTERRUPT CONTROLLER:
01330 ;
01340 KBINT2: MOV       AL,20H          ;SEND "EOI" COMMAND...
01350   OUT     20H,AL          ;...TO 8259 COMMAND REGISTER
01360   POP     AX              ;RESTORE ALL ALTERED REGISTERS!!
01370   POP     BX
01380   POP     DS
01390   IRET                    ;RETURN FROM INTERRUPT
01400 KBINT     ENDP
01410 ;
01420 ; SUBROUTINE TO DISPLAY A CHARACTER ON THE SCREEN.
01430 ; ENTER WITH AL = CHARACTER TO BE DISPLAYED.
01440 ; USES VIDEO INTERFACE IN BIOS.
01450 ;
01460 DISPCHAR PROC     NEAR
01470   PUSH    BX      ;SAVE BX REGISTER
01480   MOV     BX,0    ;SELECT DISPLAY PAGE 0
01490   MOV     AH,14   ;FUNCTION CODE FOR 'WRITE'
01500   INT     10H     ;CALL VIDEO DRIVER IN BIOS
01510   POP     BX      ;RESTORE BX REGISTER
01520   RET             ;RETURN TO CALLER OF 'DISPCHAR'
01530 DISPCHAR ENDP
01540 ;
01550 START     ENDP
01560 CODE      ENDS
01570   END     START
```

Fig. 5-7 (cont). Custom keyboard-support program.

major component of the program is our custom keyboard-support software. This also consists of two parts; they are KBINT, the keyboard interrupt-service routine, and KBGET, called from the main program to obtain keyboard input.

Let us look at the program in more detail. Statements 400 through 480 set the address of our own keyboard interrupt-service routine (KBINT) into the appropriate entry in the interrupt-service–routine address table. Recall that the keyboard interrrupt signal is sent to the IRQ1 input of the 8259. The 8259 has been programmed to identify this interrupt source with a type code of 09H. The correct address-table entry therefore begins at physical address 09H*4, or 00024H. Note that we disable interrupts (CLI) before altering the data in the address table. A catastrophic error could occur if an interrupt were to be received while the address table is being modified. Once the address table is modified, we program the interrupt-mask register of the 8259 to allow interrupts only from lines IRQ0 and IRQ1 (the timer and the keyboard, respectively). We then enable interrupts (STI) and enter the second part of the main program.

The second part (statements 550 through 630) is an infinite loop that calls routine KBGET to obtain characters input from the keyboard. Each character so received is echoed to the display screen by the DISPCHAR routine that we developed in the last chapter. Note the special code provided to detect the ENTER key (ASCII carriage return). This is necessary because a carriage return sent to an output device should always be followed by a line feed. If this is not done, we will find ourselves typing over the previous line of text.

If we strike a key while this loop is running, a type 09H interrupt will occur. This will cause our KBINT procedure to be activiated. As you may recall, the 8088 interrupt response will also save the address of the instruction that was executing, save the flags, and disable further interrupts. The first responsibility of KBINT is to save any additional registers that it will use in servicing the interrupt (statements 910-930. It then establishes addressability to our data segment by loading the data segment address into the DS register (statements 950-980). Although not actually necessary for this example, this is a wise precaution. In general, when we receive control at an interrupt service routine, we do not know where that control came from. We therefore cannot be certain of the contents of any register (in this case, the DS register).

KBINT now proceeds to read in the scan code of the key that was depressed and send back the acknowledge signal (statements 1000 through 1080). If the scan code indicates that a key was being released (bit 7 = 1), then no further action is taken (statements 1130 and 1140). Otherwise, the XLATB instruction is used to convert the scan code into its corresponding ASCII character. The XLATB instruction requires that BX point to a

translation table in the data segment. We therefore load BX with the offset address of SCANTABLE, which we have defined in our data segment. For each keyboard scan code that we wish to acknowledge, we have placed the appropriate ASCII code value into the corresponding position in SCANTABLE. Scan codes that we wish to ignore, such as those assigned to the function keys, F1–F10, are translated into a value of zero. After the translation, we test for a value of zero. If we have such a value, then the key is ignored (statements 1170 and 1180).

Assuming a valid key has been struck, we now have its ASCII code in the AL register. We must place this byte onto the circular queue so that it is available to the main program. This is accomplished by statements 1220 through 1300. The queue itself is defined in the data segment, with the name BUFFER. It has the capacity to hold up to ten keystrokes. Two pointers, named BUFPTR1 and BUFPTR2, are used to keep track of the data in the queue. They point to the beginning and end of the valid data in the queue, respectively. Data is added onto the queue by placing it at the position pointed to by BUFPTR2, and then incrementing BUFPTR2. Data is taken off the queue by removing it from the position pointed to by BUFPTR1, and then incrementing BUFPTR1. When both pointers are equal, this indicates that there is no data in the queue. When incremented past the end of the queue, each pointer "wraps around" back to the beginning of the queue. This approach, illustrated in Fig. 5-8, ensures that we always retrieve data from the queue in the same order in which it was placed onto the queue. Notice that, in our implementation, we simply ignore (lose) a character if it is received when the queue is full.

Once the data has been placed onto the queue, we complete the interrupt response by sending the "end of interrupt" signal to the 8259 (statements 1340 and 1350). We then restore all saved registers and return to the main program, at its point of interruption, via an IRET instruction.

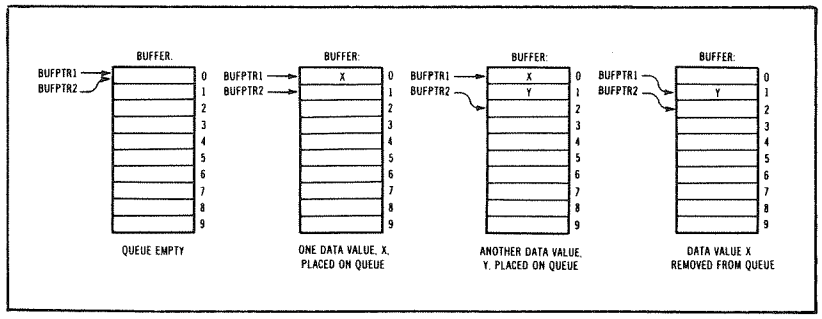

Fig. 5-8. Circular-queue operations.

The main program relies on the KBGET routine (statements 650 through 860) to retrieve keyboard data from the circular queue. This routine waits until then is some data in the queue (as indicated by BUFPTR1 not equal to BUFPTR2). It than fetches that data, advances BUFPTR1, and returns the data value to its caller. Note that we must disable interrupts while the queue pointers are being manipulated. If this is not done, a keyboard interrupt may occur while we are trying to take data off the queue. We cannot allow data to be placed onto the queue at the same time that it is being taken off the queue, because this could cause us to overlook a queue-full condition.

If you type in this program, assemble it, and run it, you will be able to type most characters on the keyboard and have them echoed on the display screen. The only control keys that will function are "Backspace" and "Enter." Most other control keys will be ignored. Most important, however, is the fact that the control-key combinations CONTROL-BREAK and CONTROL-ALT-DEL are totally disabled. These functions are normally detected by the BIOS keyboard support. Since we have not provided such detection in our own program, we have effectively "locked up" the machine; the only way to exit from our program is to *turn the machine off.* This demonstrates the power and control that an assembler-language programmer can exert over his computer.

THE 8253 TIMER

The 8253 Timer chip can perform a number of different timing and/or counting functions. Within the chip are three independent counters, numbered 0, 1, and 2. Each of these three *timer channels* can be programmed to operate in one of six different modes, referred to as mode 0 through mode 5. Once they have been programmed, all of the channels can perform their designated counting or timing operations simultaneously. As you can imagine, some very sophisticated operations can be performed with this device.

A block diagram of the 8253 is presented in Fig. 5-9. Note that the hardware related to each timer channel is identical. Each channel contains a 16-bit *latch* register and a 16-bit *counter* register. Each channel also has two dedicated input signals, called *clock* and *gate,* as well as an output signal, *out.* In general, we program a count value into the latch register. From there, it is transferred into the counter register. Each time a pulse appears on the clock input, the value in the counter register is decremented by one. When the counter register reaches zero, a signal is generated on the out line. The mode to which we program the timer channel will determine exactly how each of these operations takes place.

The 8253 is programmed by writing commands into its one-byte–wide *command register.* In addition, each channel has a dedicated, one-byte–