

The PC's keyboard is the primary human input device on the system. Although it seems rather mundane, the keyboard is the primary input device for most software, so learning how to program the keyboard properly is very important to application developers.

IBM and countless keyboard manufacturers have produced numerous keyboards for PCs and compatibles. Most modern keyboards provide at least 101 different keys and are reasonably compatible with the IBM PC/AT 101 Key Enhanced Keyboard. Those that do provide extra keys generally program those keys to emit a sequence of other keystrokes or allow the user to program a sequence of keystrokes on the extra keys. Since the 101 key keyboard is ubiquitous, we will assume its use in this chapter.

When IBM first developed the PC, they used a very simple interface between the keyboard and the computer. When IBM introduced the PC/AT, they completely redesigned the keyboard interface. Since the introduction of the PC/AT, almost every keyboard has conformed to the PC/AT standard. Even when IBM introduced the PS/2 systems, the changes to the keyboard interface were minor and upwards compatible with the PC/AT design. Therefore, this chapter will also limit its attention to PC/AT compatible devices since so few PC/XT keyboards and systems are still in use.

There are five main components to the keyboard we will consider in this chapter - basic keyboard information, the DOS interface, the BIOS interface, the int 9 keyboard interrupt service routine, and the hardware interface to the keyboard. The last section of this chapter will discuss how to fake keyboard input into an application.

---

## 20.1 Keyboard Basics

The PC's keyboard is a computer system in its own right. Buried inside the keyboard's case is an 8042 microcontroller chip that constantly scans the switches on the keyboard to see if any keys are down. This processing goes on in parallel with the normal activities of the PC, hence the keyboard never misses a keystroke because the 80x86 in the PC is busy.

A typical keystroke starts with the user pressing a key on the keyboard. This closes an electrical contact in the switch so the microcontroller can sense that you've pressed the switch. Alas, switches (being the mechanical things that they are) do not always close (make contact) so cleanly. Often, the contacts bounce off one another several times before coming to rest making a solid contact. If the microcontroller chip reads the switch constantly, these bouncing contacts will look like a very quick series of key presses and releases. This could generate *multiple* keystrokes to the main computers, a phenomenon known as *keybounce*, common to many cheap and old keyboards. But even on the most expensive and newest keyboards, keybounce is a problem if you look at the switch a million times a second; mechanical switches simply cannot settle down that quickly. Most keyboard scanning algorithms, therefore, control how often they scan the keyboard. A typical inexpensive key will settle down within five milliseconds, so if the keyboard scanning software only looks at the key every ten milliseconds, or so, the controller will effectively miss the keybounce<sup>1</sup>.

Simply noting that a key is pressed is not sufficient reason to generate a key code. A user may hold a key down for many tens of milliseconds before releasing it. The keyboard controller must not generate a new key sequence every time it scans the keyboard and finds a key held down. Instead, it should generate a single key code value when the key goes from an up position to the down position (a *down key* operation). Upon detecting a down key stroke, the microcontroller sends a keyboard *scan code* to the PC. The scan code is *not* related to the ASCII code for that key, it is an arbitrary value IBM chose when they first developed the PC's keyboard.

---

1. A typical user cannot type 100 characters/sec nor reliably press a key for less than 1/50th of a second, so scanning the keyboard at 10 msec intervals will not lose any keystrokes.

The PC keyboard actually generates *two* scan codes for every key you press. It generates a *down code* when you press a key and an *up code* when you release the key. The 8042 microcontroller chip transmits these scan codes to the PC where they are processed by the keyboard's interrupt service routine. Having separate up and down codes is important because certain keys (like shift, control, and alt) are only meaningful when held down. By generating up codes for all the keys, the keyboard ensures that the keyboard interrupt service routine knows which keys are pressed while the user is holding down one of these *modifier* keys. The following table lists the scan codes that the keyboard microcontroller transmits to the PC:

**Table 72: PC Keyboard Scan Codes (in hex)**

Key	Down	Up	Key	Down	Up	Key	Down	Up	Key	Down	Up
Esc	1	81	{	1A	9A	, <	33	B3	<i>center</i>	4C	CC
1 !	2	82	}	1B	9B	. >	34	B4	<i>right</i>	4D	CD
2 @	3	83	Enter	1C	9C	/ ?	35	B5	<i>*</i>	4E	CE
3 #	4	84	Ctrl	1D	9D	R shift	36	B6	<i>end</i>	4F	CF
4 \$	5	85	A	1E	9E	* PrtSc	37	B7	<i>down</i>	50	D0
5 %	6	86	S	1F	9F	alt	38	B8	<i>pgdn</i>	51	D1
6 ^	7	87	D	20	A0	space	39	B9	<i>ins</i>	52	D2
7 &	8	88	F	21	A1	CAPS	3A	BA	<i>del</i>	53	D3
8 *	9	89	G	22	A2	F1	3B	BB	/	E0 35	B5
9 (	0A	8A	H	23	A3	F2	3C	BC	<i>enter</i>	E0 1C	9C
0 )	0B	8B	J	24	A4	F3	3D	BD	F11	57	D7
- _	0C	8C	K	25	A5	F4	3E	BE	F12	58	D8
= +	0D	8D	L	26	A6	F5	3F	BF	ins	E0 52	D2
Bksp	0E	8E	;	27	A7	F6	40	C0	del	E0 53	D3
Tab	0F	8F	‘ ‘	28	A8	F7	41	C1	home	E0 47	C7
Q	10	90	` ~	29	A9	F8	42	C2	end	E0 4F	CF
W	11	91	L shift	2A	AA	F9	43	C3	pgup	E0 49	C9
E	12	92	\	2B	AB	F10	44	C4	pgdn	E0 51	D1
R	13	93	Z	2C	AC	NUM	45	C5	left	E0 4B	CB
T	14	94	X	2D	AD	SCRL	46	C6	right	E0 4D	CD
Y	15	95	C	2E	AE	<i>home</i>	47	C7	up	E0 48	C8
U	16	96	V	2F	AF	<i>up</i>	48	C8	down	E0 50	D0
I	17	97	B	30	B0	<i>pgup</i>	49	C9	R alt	E0 38	B8
O	18	98	N	31	B1	-	4A	CA	R ctrl	E0 1D	9D
P	19	99	M	32	B2	<i>left</i>	4B	CB	Pause	E1 1D 45 E1 9D C5	-

The keys in italics are found on the numeric keypad. Note that certain keys transmit two or more scan codes to the system. The keys that transmit more than one scan code were new keys added to the keyboard when IBM designed the 101 key enhanced keyboard.

When the scan code arrives at the PC, a second microcontroller chip receives the scan code, does a conversion on the scan code<sup>2</sup>, makes the scan code available at I/O port 60h, and then interrupts the processor and leaves it up to the keyboard ISR to fetch the scan code from the I/O port.

The keyboard (int 9) interrupt service routine reads the scan code from the keyboard input port and processes the scan code as appropriate. Note that the scan code the system receives from the keyboard microcontroller is a single value, even though some keys on the keyboard represent up to four different values. For example, the “A” key on the keyboard can produce A, a, ctrl-A, or alt-A. The actual code the system yields depends upon the current state of the modifier keys (shift, ctrl, alt, capslock, and numlock). For example, if an A key scan code comes along (1Eh) and the shift key is down, the system produces the ASCII code for an uppercase A. If the user is pressing *multiple* modifier keys the system prioritizes them from low to high as follows:

- No modifier key down
- Numlock/Capslock (same precedence, lowest priority)
- shift
- ctrl
- alt (highest priority)

Numlock and capslock affect different sets of keys<sup>3</sup>, so there is no ambiguity resulting from their equal precedence in the above chart. If the user is pressing two modifier keys at the same time, the system only recognizes the modifier key with the highest priority above. For example, if the user is pressing the ctrl and alt keys at the same time, the system only recognizes the alt key. The numlock, capslock, and shift keys are a special case. If numlock or capslock is active, pressing the shift key makes it inactive. Likewise, if numlock or capslock is inactive, pressing the shift key effectively “activates” these modifiers.

Not all modifiers are legal for every key. For example, ctrl-8 is not a legal combination. The keyboard interrupt service routine ignores all keypresses combined with illegal modifier keys. For some unknown reason, IBM decided to make certain key combinations legal and others illegal. For example, ctrl-left and ctrl-right are legal, but ctrl-up and ctrl-down are not. You’ll see how to fix this problem a little later.

The shift, ctrl, and alt keys are *active* modifiers. That is, modification to a keypress occurs only while the user holds down one of these modifier keys. The keyboard ISR keeps track of whether these keys are down or up by setting an associated bit upon receiving the down code and clearing that bit upon receiving the up code for shift, ctrl, or alt. In contrast, the numlock, scroll lock, and capslock keys are *toggle* modifiers<sup>4</sup>. The keyboard ISR inverts an associated bit every time it sees a down code followed by an up code for these keys.

Most of the keys on the PC’s keyboard correspond to ASCII characters. When the keyboard ISR encounters such a character, it translates it to a 16 bit value whose L.O. byte is the ASCII code and the H.O. byte is the key’s scan code. For example, pressing the “A” key with no modifier, with shift, and with control produces 1E61h, 1E41h, and 1E01h, respectively (“a”, “A”, and ctrl-A). Many key sequences do not have corresponding ASCII codes. For example, the function keys, the cursor control keys, and the alt key sequences do not have corresponding ASCII codes. For these special *extended* code, the keyboard ISR stores a zero in the L.O. byte (where the ASCII code typically goes) and the extended code goes in the H.O. byte. The extended code is usually, though certainly not always, the scan code for that key.

The only problem with this extended code approach is that the value zero is a legal ASCII character (the NUL character). Therefore, you cannot directly enter NUL characters into an application. If an application must input NUL characters, IBM has set aside the extended code 0300h (ctrl-3) for this purpose. Your application must explicitly convert this extended code to the NUL character (actually, it need only recog-

---

2. The keyboard doesn’t actually transmit the scan codes appearing in the previous table. Instead, it transmits its own scan code that the PC’s microcontroller translates to the scan codes in the table. Since the programmer never sees the native scan codes so we will ignore them.

3. Numlock only affects the keys on the numeric keypad, capslock only affects the alphabetic keys.

4. It turns out the INS key is also a toggle modifier, since it toggles a bit in the BIOS variable area. However, INS also returns a scan code, the other modifiers do not.

nize the H.O. value 03, since the L.O. byte already is the NUL character). Fortunately, very few programs need to allow the input of the NUL character from the keyboard, so this problem is rarely an issue.

The following table lists the scan and extended key codes the keyboard ISR generates for applications in response to a keypress with various modifiers. Extended codes are in italics. All other values (except the scan code column) represent the L.O. eight bits of the 16 bit code. The H.O. byte comes from the scan code column.

**Table 73: Keyboard Codes (in hex)**

Key	Scan Code	ASCII	Shift <sup>a</sup>	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num
Esc	01	1B	1B	1B		1B	1B	1B	1B
1 !	02	31	21		<i>7800</i>	31	31	31	31
2 @	03	32	40	<i>0300</i>	<i>7900</i>	32	32	32	32
3 #	04	33	23		<i>7A00</i>	33	33	33	33
4 \$	05	34	24		<i>7B00</i>	34	34	34	34
5 %	06	35	25		<i>7C00</i>	35	35	35	35
6 ^	07	36	5E	1E	<i>7D00</i>	36	36	36	36
7 &	08	37	26		<i>7E00</i>	37	37	37	37
8 *	09	38	2A		<i>7F00</i>	38	38	38	38
9 (	0A	39	28		<i>8000</i>	39	39	39	39
0 )	0B	30	29		<i>8100</i>	30	30	30	30
- _	0C	2D	5F	1F	<i>8200</i>	2D	2D	5F	5F
= +	0D	3D	2B		<i>8300</i>	3D	3D	2B	2B
Bksp	0E	08	08	7F		08	08	08	08
Tab	0F	09	<i>0F00</i>			09	09	<i>0F00</i>	<i>0F00</i>
Q	10	71	51	11	<i>1000</i>	71	51	71	51
W	11	77	57	17	<i>1100</i>	77	57	77	57
E	12	65	45	05	<i>1200</i>	65	45	65	45
R	13	72	52	12	<i>1300</i>	72	52	72	52
T	14	74	54	14	<i>1400</i>	74	54	74	54
Y	15	79	59	19	<i>1500</i>	79	59	79	59
U	16	75	55	15	<i>1600</i>	75	55	75	55
I	17	69	49	09	<i>1700</i>	69	49	69	49
O	18	6F	4F	0F	<i>1800</i>	6F	4F	6F	4F
P	19	70	50	10	<i>1900</i>	70	50	70	50
[ {	1A	5B	7B	1B		5B	5B	7B	7B
] }	1B	5D	7D	1D		5D	5D	7D	7D
enter	1C	0D	0D	0A		0D	0D	0A	0A
ctrl	1D								
A	1E	61	41	01	<i>1E00</i>	61	41	61	41
S	1F	73	53	13	<i>1F00</i>	73	53	73	53
D	20	64	44	04	<i>2000</i>	64	44	64	44
F	21	66	46	06	<i>2100</i>	66	46	66	46
G	22	67	47	07	<i>2200</i>	67	47	67	47
H	23	68	48	08	<i>2300</i>	68	48	68	48
J	24	6A	4A	0A	<i>2400</i>	6A	4A	6A	4A
K	25	6B	4B	0B	<i>2500</i>	6B	4B	6B	4B
L	26	6C	4C	0C	<i>2600</i>	6C	4C	6C	4C
; :	27	3B	3A			3B	3B	3A	3A
' "	28	27	22			27	27	22	22
Key	Scan Code	ASCII	Shift	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num

Table 73: Keyboard Codes (in hex)

Key	Scan Code	ASCII	Shift <sup>a</sup>	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num
~	29	60	7E			60	60	7E	7E
Lshift	2A								
\	2B	5C	7C	1C		5C	5C	7C	7C
Z	2C	7A	5A	1A	<b>2C00</b>	7A	5A	7A	5A
X	2D	78	58	18	<b>2D00</b>	78	58	78	58
C	2E	63	43	03	<b>2E00</b>	63	43	63	43
V	2F	76	56	16	<b>2F00</b>	76	56	76	56
B	30	62	42	02	<b>3000</b>	62	42	62	42
N	31	6E	4E	0E	<b>3100</b>	6E	4E	6E	4E
M	32	6D	4D	0D	<b>3200</b>	6D	4D	6D	4D
, <	33	2C	3C			2C	2C	3C	3C
. >	34	2E	3E			2E	2E	3E	3E
/ ?	35	2F	3F			2F	2F	3F	3F
Rshift	36								
* PrtSc	37	2A	INT 5 <sup>b</sup>	10 <sup>c</sup>		2A	2A	INT 5	INT 5
alt	38								
space	39	20	20	20		20	20	20	20
caps	3A								
F1	3B	<b>3B00</b>	<b>5400</b>	<b>5E00</b>	<b>6800</b>	<b>3B00</b>	<b>3B00</b>	<b>5400</b>	<b>5400</b>
F2	3C	<b>3C00</b>	<b>5500</b>	<b>5F00</b>	<b>6900</b>	<b>3C00</b>	<b>3C00</b>	<b>5500</b>	<b>5500</b>
F3	3D	<b>3D00</b>	<b>5600</b>	<b>6000</b>	<b>6A00</b>	<b>3D00</b>	<b>3D00</b>	<b>5600</b>	<b>5600</b>
F4	3E	<b>3E00</b>	<b>5700</b>	<b>6100</b>	<b>6B00</b>	<b>3E00</b>	<b>3E00</b>	<b>5700</b>	<b>5700</b>
F5	3F	<b>3F00</b>	<b>5800</b>	<b>6200</b>	<b>6C00</b>	<b>3F00</b>	<b>3F00</b>	<b>5800</b>	<b>5800</b>
F6	40	<b>4000</b>	<b>5900</b>	<b>6300</b>	<b>6D00</b>	<b>4000</b>	<b>4000</b>	<b>5900</b>	<b>5900</b>
F7	41	<b>4100</b>	<b>5A00</b>	<b>6400</b>	<b>6E00</b>	<b>4100</b>	<b>4100</b>	<b>5A00</b>	<b>5A00</b>
F8	42	<b>4200</b>	<b>5B00</b>	<b>6500</b>	<b>6F00</b>	<b>4200</b>	<b>4200</b>	<b>5B00</b>	<b>5B00</b>
F9	43	<b>4300</b>	<b>5C00</b>	<b>6600</b>	<b>7000</b>	<b>4300</b>	<b>4300</b>	<b>5C00</b>	<b>5C00</b>
F10	44	<b>4400</b>	<b>5D00</b>	<b>6700</b>	<b>7100</b>	<b>4400</b>	<b>4400</b>	<b>5D00</b>	<b>5D00</b>
num	45								
scrl	46								
home	47	<b>4700</b>	37	<b>7700</b>		37	4700	37	4700
up	48	<b>4800</b>	38			38	4800	38	4800
pgup	49	<b>4900</b>	39	<b>8A00</b>		39	4900	39	4900
_ <sup>d</sup>	4A	2D	2D			2D	2D	2D	2D
left	4B	<b>4B00</b>	34	<b>7300</b>		34	4B00	34	4B00
center	4C	<b>4C00</b>	35			35	4C00	35	4C00
right	4D	<b>4D00</b>	36	<b>7400</b>		36	4D00	36	4D00
+ <sup>e</sup>	4E	2B	2B			2B	2B	2B	2B
end	4F	<b>4F00</b>	31	<b>7500</b>		31	4F00	31	4F00
down	50	<b>5000</b>	32			32	5000	32	5000
pgdn	51	<b>5100</b>	33	<b>7600</b>		33	5100	33	5100
ins	52	<b>5200</b>	30			30	5200	30	5200
del	53	<b>5300</b>	2E			2E	5300	2E	5300
Key	Scan Code	ASCII	Shift	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num

a. For the alphabetic characters, if capslock is active then see the shift-capslock column.

b. Pressing the PrtSc key does not produce a scan code. Instead, BIOS executes an int 5 instruction which should print the screen.

c. This is the control-P character that will activate the printer under MS-DOS.

d. This is the minus key on the keypad.

e. This is the plus key on the keypad.

The 101-key keyboards generally provide an enter key and a “/” key on the numeric keypad. Unless you write your own int 9 keyboard ISR, you will not be able to differentiate these keys from the ones on the main keyboard. The separate cursor control pad also generates the same extended codes as the numeric keypad, except it never generates numeric ASCII codes. Otherwise, you cannot differentiate these keys from the equivalent keys on the numeric keypad (assuming numlock is off, of course).

The keyboard ISR provides a special facility that lets you enter the ASCII code for a keystroke directly from the keyboard. To do this, hold down the alt key and typing out the *decimal* ASCII code (0..255) for a character on the numeric keypad. The keyboard ISR will convert these keystrokes to an eight-bit value, attach an H.O. byte of zero to the character, and use that as the character code.

The keyboard ISR inserts the 16 bit value into the PC's *type ahead buffer*. The system type ahead buffer is a circular queue that uses the following variables

```
40:1A - HeadPtr word ?
40:1C - TailPtr word ?
40:1E - Buffer word 16 dup (?)
```

The keyboard ISR inserts data at the location pointed at by TailPtr. The BIOS keyboard function removes characters from the location pointed at by the HeadPtr variable. These two pointers almost always contain an offset into the Buffer array<sup>5</sup>. If these two pointers are equal, the type ahead buffer is empty. If the value in HeadPtr is two greater than the value in TailPtr (or HeadPtr is 1Eh and TailPtr is 3Ch), then the buffer is full and the keyboard ISR will reject any additional keystrokes.

Note that the TailPtr variable always points at the next available location in the type ahead buffer. Since there is no “count” variable providing the number of entries in the buffer, we must always leave one entry free in the buffer area; this means the type ahead buffer can only hold 15 keystrokes, not 16.

In addition to the type ahead buffer, the BIOS maintains several other keyboard-related variables in segment 40h. The following table lists these variables and their contents:

**Table 74: Keyboard Related BIOS Variables**

Name	Address <sup>a</sup>	Size	Description
KbdFlags1 (modifier flags)	40:17	Byte	This byte maintains the current status of the modifier keys on the keyboard. The bits have the following meanings: bit 7: Insert mode toggle bit 6: Capslock toggle (1=capslock on) bit 5: Numlock toggle (1=numlock on) bit 4: Scroll lock toggle (1=scroll lock on) bit 3: Alt key (1=alt is down) bit 2: Ctrl key (1=ctrl is down) bit 1: Left shift key (1=left shift is down) bit 0: Right shift key (1=right shift is down)

5. It is possible to change these pointers so they point elsewhere in the 40H segment, but this is not a good idea because many applications assume that these two pointers contain a value in the range 1Eh..3Ch.

**Table 74: Keyboard Related BIOS Variables**

Name	Address <sup>a</sup>	Size	Description
KbdFlags2 (Toggle keys down)	40:18	Byte	Specifies if a toggle key is currently down. bit 7: Insert key (currently down if 1) bit 6: Capslock key (currently down if 1) bit 5: Numlock key (currently down if 1) bit 4: Scroll lock key (currently down if 1) bit 3: Pause state locked (ctrl-Numlock) if one bit 2: SysReq key (currently down if 1) bit 1: Left alt key (currently down if 1) bit 0: Left ctrl key (currently down if 1)
AltKpd	40:19	Byte	BIOS uses this to compute the ASCII code for an alt-Keypad sequence.
BufStart	40:80	Word	Offset of start of keyboard buffer (1Eh). Note: this variable is not supported on many systems, be careful if you use it.
BufEnd	40:82	Word	Offset of end of keyboard buffer (3Eh). See the note above.
KbdFlags3	40:96	Byte	Miscellaneous keyboard flags. bit 7: Read of keyboard ID in progress bit 6: Last char is first kbd ID character bit 5: Force numlock on reset bit 4: 1 if 101-key kbd, 0 if 83/84 key kbd. bit 3: Right alt key pressed if 1 bit 2: Right ctrl key pressed if 1 bit 1: Last scan code was E0h bit 0: Last scan code was E1h
KbdFlags4	40:97	Byte	More miscellaneous keyboard flags. bit 7: Keyboard transmit error bit 6: Mode indicator update bit 5: Resend receive flag bit 4: Acknowledge received bit 3: Must always be zero bit 2: Capslock LED (1=on) bit 1: Numlock LED (1=on) bit 0: Scroll lock LED (1=on)

a. Addresses are all given in hexadecimal

One comment is in order about KbdFlags1 and KbdFlags4. Bits zero through two of the KbdFlags4 variable is BIOS' current settings for the LEDs on the keyboard. periodically, BIOS compares the values for capslock, numlock, and scroll lock in KbdFlags1 against these three bits in KbdFlags4. If they do not agree, BIOS will send an appropriate command to the keyboard to update the LEDs and it will change the values in the KbdFlags4 variable so the system is consistent. Therefore, if you mask in new values for numlock, scroll lock, or caps lock, the BIOS will automatically adjust KbdFlags4 and set the LEDs accordingly.

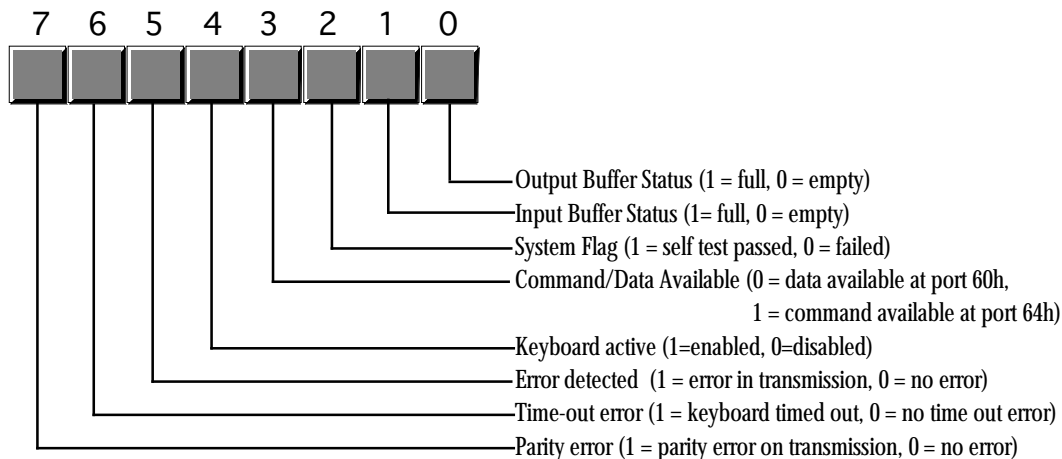
---

## 20.2 The Keyboard Hardware Interface

IBM used a very simple hardware design for the keyboard port on the original PC and PC/XT machines. When they introduced the PC/AT, IBM completely resigned the interface between the PC and

the keyboard. Since then, almost every PC model and PC clone has followed this keyboard interface standard<sup>6</sup>. Although IBM extended the capabilities of the keyboard controller when they introduced their PS/2 systems, the PS/2 models are still upwards compatible from the PC/AT design. Since there are so few original PCs in use today (and fewer people write original software for them), we will ignore the original PC keyboard interface and concentrate on the AT and later designs.

There are two keyboard microcontrollers that the system communicates with – one on the PC's motherboard (the *on-board* microcontroller) and one inside the keyboard case (the *keyboard* microcontroller). Communication with the on-board microcontroller is through I/O port 64h. Reading this byte provides the status of the keyboard controller. Writing to this byte sends the on-board microcontroller a command. The organization of the status byte is



**On-Board 8042 Keyboard Microcontroller Status Byte (Read Port 64h)**

Communication to the microcontroller in the keyboard unit is via the bytes at I/O addresses 60h and 64h. Bits zero and one in the status byte at port 64h provide the necessary *handshaking* control for these ports. Before writing any data to these ports, bit zero of port 64h must be zero; data is available for reading from port 60h when bit one of port 64h contains a one. The keyboard enable and disable bits in the command byte (port 64h) determine whether the keyboard is active and whether the keyboard will interrupt the system when the user presses (or releases) a key, etc.

Bytes written to port 60h are sent to the keyboard microcontroller and bytes written to port 64h are sent to the on-board microcontroller. Bytes read from port 60h generally come from the keyboard, although you can program the on-board microcontroller to return certain values at this port, as well. The following tables lists the commands sent to the keyboard microcontroller and the values you can expect back. The following table lists the allowable commands you can write to port 64h:

**Table 75: On-Board Keyboard Controller Commands (Port 64h)**

Value (hex)	Description
20	Transmit keyboard controller's command byte to system as a scan code at port 60h.
60	The next byte written to port 60h will be stored in the keyboard controller's command byte.

6. We will ignore the PCjr machine in this discussion.



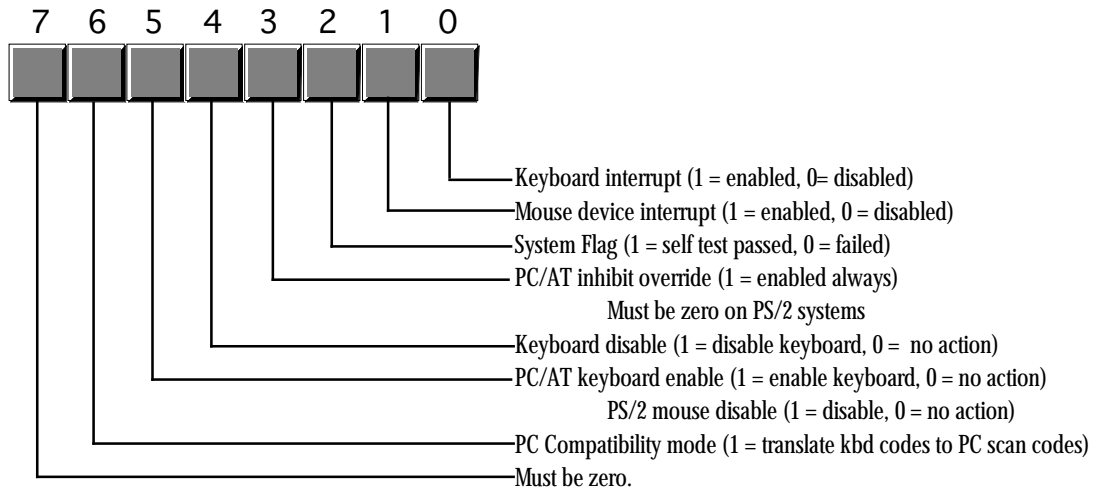
**Table 75: On-Board Keyboard Controller Commands (Port 64h)**

Value (hex)	Description
A4	Test if a password is installed (PS/2 only). Result comes back in port 60h. 0FAh means a password is installed, 0F1h means no password.
A5	Transmit password (PS/2 only). Starts receipt of password. The next sequence of scan codes written to port 60h, ending with a zero byte, are the new password.
A6	Password match. Characters from the keyboard are compared to password until a match occurs.
A7	Disable mouse device (PS/2 only). Identical to setting bit five of the command byte.
A8	Enable mouse device (PS/2 only). Identical to clearing bit five of the command byte.
A9	Test mouse device. Returns 0 if okay, 1 or 2 if there is a stuck clock, 3 or 4 if there is a stuck data line. Results come back in port 60h.
AA	Initiates self-test. Returns 55h in port 60h if successful.
AB	Keyboard interface test. Tests the keyboard interface. Returns 0 if okay, 1 or 2 if there is a stuck clock, 3 or 4 if there is a stuck data line. Results come back in port 60h.
AC	Diagnostic. Returns 16 bytes from the keyboard's microcontroller chip. Not available on PS/2 systems.
AD	Disable keyboard. Same operation as setting bit four of the command register.
AE	Enable keyboard. Same operation as clearing bit four of the command register.
C0	Read keyboard input port to port 60h. This input port contains the following values: bit 7: Keyboard inhibit keyswitch (0 = inhibit, 1 = enabled). bit 6: Display switch (0=color, 1=mono). bit 5: Manufacturing jumper. bit 4: System board RAM (always 1). bits 0-3: undefined.
C1	Copy input port (above) bits 0-3 to status bits 4-7. (PS/2 only)
C2	Copy input port (above) bits 4-7 to status port bits 4-7. (PS/2 only).
D0	Copy microcontroller output port value to port 60h (see definition below).
D1	Write the next data byte written to port 60h to the microcontroller output port. This port has the following definition: bit 7: Keyboard data. bit 6: Keyboard clock. bit 5: Input buffer empty flag. bit 4: Output buffer full flag. bit 3: Undefined. bit 2: Undefined. bit 1: Gate A20 line. bit 0: System reset (if zero).  Note: writing a zero to bit zero will reset the machine. Writing a one to bit one combines address lines 19 and 20 on the PC's address bus.
D2	Write keyboard buffer. The keyboard controller returns the next value sent to port 60h as though a keypress produced that value. (PS/2 only).
D3	Write mouse buffer. The keyboard controller returns the next value sent to port 60h as though a mouse operation produced that value. (PS/2 only).
D4	Writes the next data byte (60h) to the mouse (auxiliary) device. (PS/2 only).

**Table 75: On-Board Keyboard Controller Commands (Port 64h)**

Value (hex)	Description
E0	Read test inputs. Returns in port 60h the status of the keyboard serial lines. Bit zero contains the keyboard clock input, bit one contains the keyboard data input.
F <sub>x</sub>	Pulse output port (see definition for D1). Bits 0-3 of the keyboard controller command byte are pulsed onto the output port. Resets the system if bit zero is a zero.

Commands 20h and 60h let you read and write the *keyboard controller command byte*. This byte is internal to the on-board microcontroller and has the following layout:



### On-Board 8042 Keyboard Microcontroller Command byte (see commands 20h and 60h)

The system transmits bytes written to I/O port 60h directly to the keyboard's microcontroller. Bit zero of the status register must contain a zero before writing any data to this port. The commands the keyboard recognizes are

**Table 76: Keyboard Microcontroller Commands (Port 60h)**

Value (hex)	Description
ED	Send LED bits. The next byte written to port 60h updates the LEDs on the keyboard. The parameter (next) byte contains: bits 3-7: Must be zero. bit 2: Capslock LED (1 = on, 0 = off). bit 1: Numlock LED (1 = on, 0 = off). bit 0: Scroll lock LED (1 = on, 0 = off).
EE	Echo commands. Returns 0EEh in port 60h as a diagnostic aid.

**Table 76: Keyboard Microcontroller Commands (Port 60h)**

Value (hex)	Description
F0	Select alternate scan code set (PS/2 only). The next byte written to port 60h selects one of the following options: 00: Report current scan code set in use (next value read from port 60h). 01: Select scan code set #1 (standard PC/AT scan code set). 02: Select scan code set #2. 03: Select scan code set #3.
F2	Send two-byte keyboard ID code as the next two bytes read from port 60h (PS/2 only).
F3	Set Autorepeat delay and repeat rate. Next byte written to port 60h determines rate: bit 7: must be zero bits 5,6: Delay. 00- 1/4 sec, 01- 1/2 sec, 10- 3/4 sec, 11- 1 sec. bits 0-4: Repeat rate. 0- approx 30 chars/sec to 1Fh- approx 2 chars/sec.
F4	Enable keyboard.
F5	Reset to power on condition and wait for enable command.
F6	Reset to power on condition and begin scanning keyboard.
F7	Make all keys autorepeat (PS/2 only).
F8	Set all keys to generate an up code and a down code (PS/2 only).
F9	Set all keys to generate an up code only (PS/2 only).
FA	Set all keys to autorepeat and generate up and down codes (PS/2 only).
FB	Set an individual key to autorepeat. Next byte contains the scan code of the desired key. (PS/2 only).
FC	Set an individual key to generate up and down codes. Next byte contains the scan code of the desired key. (PS/2 only).
FD	Set an individual key to generate only down codes. Next byte contains the scan code of the desired key. (PS/2 only).
FE	Resend last result. Use this command if there is an error receiving data.
FF	Reset keyboard to power on state and start the self-test.

The following short program demonstrates how to send commands to the keyboard's controller. This little TSR utility programs a "light show" on the keyboard's LEDs.

```

; LEDSHOW.ASM
;
; This short TSR creates a light show on the keyboard's LEDs. For space
; reasons, this code does not implement a multiplex handler nor can you
; remove this TSR once installed. See the chapter on resident programs
; for details on how to do this.
;
; cseg and EndResident must occur before the standard library segments!

cseg          segment      para public 'code'
ends

; Marker segment, to find the end of the resident section.

EndResident  segment      para public 'Resident'
EndResident  ends

               .xlist
               include     stdlib.a
               includelib stdlib.lib
               .list

```

```

byp          equ          <byte ptr>

cseg         segment      para public 'code'
              assume      cs:cseg, ds:cseg

; SetCmd-    Sends the command byte in the AL register to the 8042
;            keyboard microcontroller chip (command register at
;            port 64h).

SetCmd       proc          near
              push        cx
              push        ax          ;Save command value.
              cli          ;Critical region, no ints now.

; Wait until the 8042 is done processing the current command.

Wait4Empty:  xor          cx, cx          ;Allow 65,536 times thru loop.
              in          al, 64h       ;Read keyboard status register.
              test        al, 10b       ;Input buffer full?
              loopnz      Wait4Empty    ;If so, wait until empty.

; Okay, send the command to the 8042:

              pop         ax           ;Retrieve command.
              out         64h, al
              sti          ;Okay, ints can happen again.
              pop         cx
              ret
SetCmd       endp

; SendCmd-   The following routine sends a command or data byte to the
;            keyboard data port (port 60h).

SendCmd      proc          near
              push        ds
              push        bx
              push        cx
              mov         cx, 40h
              mov         ds, cx
              mov         bx, ax        ;Save data byte

              mov         al, 0ADh      ;Disable kbd for now.
              call        SetCmd

              cli          ;Disable ints while accessing HW.

; Wait until the 8042 is done processing the current command.

Wait4Empty:  xor          cx, cx          ;Allow 65,536 times thru loop.
              in          al, 64h       ;Read keyboard status register.
              test        al, 10b       ;Input buffer full?
              loopnz      Wait4Empty    ;If so, wait until empty.

; Okay, send the data to port 60h

              mov         al, bl
              out         60h, al

              mov         al, 0AEh      ;Reenable keyboard.
              call        SetCmd
              sti          ;Allow interrupts now.

              pop         cx
              pop         bx
              pop         ds
              ret
SendCmd      endp

```

```

; SetLEDS-      Writes the value in AL to the LEDs on the keyboard.
;              Bits 0..2 correspond to scroll, num, and caps lock,
;              respectively.

SetLEDS        proc      near
               push     ax
               push     cx

               mov     ah, al          ;Save LED bits.

               mov     al, 0EDh       ;8042 set LEDs cmd.
               call    SendCmd        ;Send the command to 8042.
               mov     al, ah         ;Get parameter byte
               call    SendCmd        ;Send parameter to the 8042.

               pop     cx
               pop     ax
               ret
SetLEDS        endp

; MyInt1C-      Every 1/4 seconds (every 4th call) this routine
;              rotates the LEDs to produce an interesting light show.

CallsPerIter   equ      4
CallCnt        byte    CallsPerIter
LEDIndex       word    LEDTable
LEDTable       byte    111b, 110b, 101b, 011b, 111b, 110b, 101b, 011b
               byte    111b, 110b, 101b, 011b, 111b, 110b, 101b, 011b
               byte    111b, 110b, 101b, 011b, 111b, 110b, 101b, 011b
               byte    111b, 110b, 101b, 011b, 111b, 110b, 101b, 011b

               byte    000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b
               byte    000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b
               byte    000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b
               byte    000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b

               byte    000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b
               byte    000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b
               byte    000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b
               byte    000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b

               byte    010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b
               byte    010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b
               byte    010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b
               byte    010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b

               byte    000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
               byte    000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
               byte    000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
               byte    000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
TableEnd       equ      this byte

OldInt1C       dword    ?

MyInt1C        proc      far
               assume   ds:cseg

               push    ds
               push    ax
               push    bx

               mov     ax, cs
               mov     ds, ax

               dec     CallCnt
               jne     NotYet
               mov     CallCnt, CallsPerIter          ;Reset call count.
               mov     bx, LEDIndex
               mov     al, [bx]
               call    SetLEDS

```

```

        inc      bx
        cmp     bx, offset TableEnd
        jne    SetTbl
        lea    bx, LEDTable
SetTbl:  mov     LEDIndex, bx
NotYet:  pop     bx
        pop     ax
        pop     ds
        jmp    cs:OldInt1C
MyInt1C endp

Main    proc

        mov     ax, cseg
        mov     ds, ax

        print
        byte   "LED Light Show",cr,lf
        byte   "Installing...",cr,lf,0

; Patch into the INT 1Ch interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 1Ch values directly into
; the OldInt1C variable.

        cli                                ;Turn off interrupts!
        mov     ax, 0
        mov     es, ax
        mov     ax, es:[1Ch*4]
        mov     word ptr OldInt1C, ax
        mov     ax, es:[1Ch*4 + 2]
        mov     word ptr OldInt1C+2, ax
        mov     es:[1Ch*4], offset MyInt1C
        mov     es:[1Ch*4+2], cs
        sti                                ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

        print
        byte   "Installed.",cr,lf,0

        mov     ah, 62h                    ;Get this program's PSP
        int     21h                        ; value.

        mov     dx, EndResident            ;Compute size of program.
        sub     dx, bx
        mov     ax, 3100h                  ;DOS TSR command.
        int     21h

Main    endp
cseg    ends

sseg    segment    para stack 'stack'
stk     db         1024 dup ("stack ")
sseg    ends

zzzzzzseg segment    para public 'zzzzzz'
LastBytes db         16 dup (?)
zzzzzzseg ends
end     Main

```

The keyboard microcontroller also sends data to the on-board microcontroller for processing and release to the system through port 60h. Most of these values are key press scan codes (up or down codes), but the keyboard transmits several other values as well. A well designed keyboard interrupt service routine should be able to handle (or at least ignore) the non-scan code values. Any particular, any program that sends commands to the keyboard needs to be able to handle the resend and acknowledge commands

that the keyboard microcontroller returns in port 60h. The keyboard microcontroller sends the following values to the system:

**Table 77: Keyboard to System Transmissions**

Value (hex)	Description
00	Data overrun. System sends a zero byte as the last value when the keyboard controller's internal buffer overflows.
1..58 81..D8	Scan codes for key presses. The positive values are down codes, the negative values (H.O. bit set) are up codes.
83AB	Keyboard ID code returned in response to the F2 command (PS/2 only).
AA	Returned during basic assurance test after reset. Also the up code for the left shift key.
EE	Returned by the ECHO command.
F0	Prefix to certain up codes (N/A on PS/2).
FA	Keyboard acknowledge to keyboard commands other than resend or ECHO.
FC	Basic assurance test failed (PS/2 only).
FD	Diagnostic failure (not available on PS/2).
FE	Resend. Keyboard requests the system to resend the last command.
FF	Key error (PS/2 only).

Assuming you have not disabled keyboard interrupts (see the keyboard controller command byte), any value the keyboard microcontroller sends to the system through port 60h will generate an interrupt on IRQ line one (int 9). Therefore, the keyboard interrupt service routine normally handles all the above codes. If you are patching into int 9, don't forget to send an end of interrupt (EOI) signal to the 8259A PIC at the end of your ISR code. Also, don't forget you can enable or disable the keyboard interrupt at the 8259A.

In general, your application software should *not* access the keyboard hardware directly. Doing so will probably make your software incompatible with utility software such as keyboard enhancers (keyboard macro programs), pop-up software, and other resident programs that read the keyboard or insert data into the system's type ahead buffer. Fortunately, DOS and BIOS provide an excellent set of functions to read and write keyboard data. Your programs will be much more robust if you stick to using those functions. Accessing the keyboard hardware directly should be left to keyboard ISRs and those keyboard enhancers and pop-up programs that absolutely have to talk directly to the hardware.

---

### 20.3 The Keyboard DOS Interface

MS-DOS provides several calls to read characters from the keyboard (see "MS-DOS, PC-BIOS, and File I/O" on page 699). The primary thing to note about the DOS calls is that they only return a single byte. This means that you lose the scan code information the keyboard interrupt service routine saves in the type ahead buffer.

If you press a key that has an extended code rather than an ASCII code, MS-DOS returns two keycodes. On the first call MS-DOS returns a zero value. This tells you that you must call the get character routine again. The code MS-DOS returns on the second call is the extended key code.

Note that the Standard Library routines call MS-DOS to read characters from the keyboard. Therefore, the Standard Library `getc` routine also returns extended keycodes in this manner. The `gets` and `getsm`

routines throw away any non-ASCII keystrokes since it would not be a good thing to insert zero bytes into the middle of a zero terminated string.

## 20.4 The Keyboard BIOS Interface

Although MS-DOS provides a reasonable set of routines to read ASCII and extended character codes from the keyboard, the PC's BIOS provides much better keyboard input facilities. Furthermore, there are lots of interesting keyboard related variables in the BIOS data area you can poke around at. In general, if you do not need the I/O redirection facilities provided by MS-DOS, reading your keyboard input using BIOS functions provides much more flexibility.

To call the MS-DOS BIOS keyboard services you use the int 16h instruction. The BIOS provides the following keyboard functions:

**Table 78: BIOS Keyboard Support Functions**

Function # (AH)	Input Parameters	Output Parameters	Description
0		a1- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. a1- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows:  bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down
3	a1 = 5 bh = 0, 1, 2, 3 for 1/4, 1/2, 3/4, or 1 second delay b1 = 0..1Fh for 30/sec to 2/sec.		Set auto repeat rate. The bh register contains the amount of time to wait before starting the autorepeat operation, the b1 register contains the autorepeat rate.
5	ch = scan code c1 = ASCII code		Store keycode in buffer. This function stores the value in the cx register at the end of the type ahead buffer. Note that the scan code in ch doesn't have to correspond to the ASCII code appearing in c1. This routine will simply insert the data you provide into the system type ahead buffer.



**Table 78: BIOS Keyboard Support Functions**

Function # (AH)	Input Parameters	Output Parameters	Description
10h		al- ASCII character ah- scan code	Read extended character. Like ah=0 call, except this one passes all key codes, the ah=0 call throws away codes that are not PC/XT compatible.
11h		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Like the ah=01h call except this one does not throw away keycodes that are not PC/XT compatible (i.e., the extra keys found on the 101 key keyboard).
12h		al- shift flags ah- extended shift flags	Returns the current status of the shift flags in ax. The shift flags are defined as follows:  bit 15: SysReq key pressed bit 14: Capslock key currently down bit 13: Numlock key currently down bit 12: Scroll lock key currently down bit 11: Right alt key is down bit 10: Right ctrl key is down bit 9: Left alt key is down bit 8: Left ctrl key is down bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Either alt key is down (some machines, left only) bit 2: Either ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

Note that many of these functions are not supported in every BIOS that was ever written. In fact, only the first three functions were available in the original PC. However, since the AT came along, most BIOSes have supported *at least* the functions above. Many BIOS provide extra functions, and there are many TSR applications you can buy that extend this list even farther. The following assembly code demonstrates how to write an int 16h TSR that provides all the functions above. You can easily extend this if you desire.

```

; INT16.ASM
;
; A short passive TSR that replaces the BIOS' int 16h handler.
; This routine demonstrates the function of each of the int 16h
; functions that a standard BIOS would provide.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg          segment      para public 'code'
ends

; Marker segment, to find the end of the resident section.

```

```

EndResident    segment    para public 'Resident'
EndResident    ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

byp            equ        <byte ptr>

cseg           segment    para public 'code'
                assume    cs:cseg, ds:cseg

OldInt16       dword     ?

; BIOS variables:

KbdFlags1     equ        <ds:[17h]>
KbdFlags2     equ        <ds:[18h]>
AltKpd        equ        <ds:[19h]>
HeadPtr       equ        <ds:[1ah]>
TailPtr       equ        <ds:[1ch]>
Buffer        equ        1eh
EndBuf        equ        3eh

KbdFlags3     equ        <ds:[96h]>
KbdFlags4     equ        <ds:[97h]>

incptr        macro      which
                local    NoWrap
                add      bx, 2
                cmp      bx, EndBuf
                jb       NoWrap
                mov      bx, Buffer
NoWrap:       mov      which, bx
                endm

; MyInt16-      This routine processes the int 16h function requests.
;
;              AH      Description
;              --      -----
;              00h     Get a key from the keyboard, return code in AX.
;              01h     Test for available key, ZF=1 if none, ZF=0 and
;                      AX contains next key code if key available.
;              02h     Get shift status. Returns shift key status in AL.
;              03h     Set Autorepeat rate. BH=0,1,2,3 (delay time in
;                      quarter seconds), BL=0..1Fh for 30 char/sec to
;                      2 char/sec repeat rate.
;              05h     Store scan code (in CX) in the type ahead buffer.
;              10h     Get a key (same as 00h in this implementation).
;              11h     Test for key (same as 01h).
;              12h     Get extended key status. Returns status in AX.

MyInt16       proc      far
                test     ah, 0EFh            ;Check for 0h and 10h
                je       GetKey
                cmp      ah, 2              ;Check for 01h and 02h
                jb       TestKey
                je       GetStatus
                cmp      ah, 3              ;Check for AutoRpt function.
                je       SetAutoRpt
                cmp      ah, 5              ;Check for StoreKey function.
                je       StoreKey
                cmp      ah, 11h           ;Extended test key opcode.
                je       TestKey
                cmp      ah, 12h           ;Extended status call
                je       ExtStatus

```

```

; Well, it's a function we don't know about, so just return to the caller.

```

```

        ired

; If the user specified ah=0 or ah=10h, come down here (we will not
; differentiate between extended and original PC getc calls).

GetKey:  mov     ah, 11h
        int     16h           ;See if key is available.
        je     GetKey        ;Wait for keystroke.

        push   ds
        push   bx
        mov    ax, 40h
        mov    ds, ax

        cli           ;Critical region! Ints off.
        mov    bx, HeadPtr ;Ptr to next character.
        mov    ax, [bx]   ;Get the character.
        incptr HeadPtr    ;Bump up HeadPtr
        pop    bx
        pop    ds
        ired           ;Restores interrupt flag.

; TestKey- Checks to see if a key is available in the keyboard buffer.
;           We need to turn interrupts on here (so the kbd ISR can
;           place a character in the buffer if one is pending).
;           Generally, you would want to save the interrupt flag here.
;           But BIOS always forces interrupts on, so there may be some
;           programs out there that depend on this, so we won't "fix"
;           this problem.
;
;           Returns key status in ZF and AX. If ZF=1 then no key is
;           available and the value in AX is indeterminate. If ZF=0
;           then a key is available and AX contains the scan/ASCII
;           code of the next available key. This call does not remove
;           the next character from the input buffer.

TestKey: sti           ;Turn on the interrupts.
        push   ds
        push   bx
        mov    ax, 40h
        mov    ds, ax
        cli           ;Critical region, ints off!
        mov    bx, HeadPtr
        mov    ax, [bx] ;BIOS returns avail keycode.
        cmp    bx, TailPtr ;ZF=1, if empty buffer
        pop    bx
        pop    ds
        sti           ;Inst back on.
        retf    2       ;Pop flags (ZF is important!)

; The GetStatus call simply returns the KbdFlags1 variable in AL.

GetStatus: push    ds
          mov    ax, 40h
          mov    ds, ax
          mov    al, KbdFlags1 ;Just return Std Status.
          pop    ds
          ired

; StoreKey- Inserts the value in CX into the type ahead buffer.

StoreKey: push    ds
          push   bx
          mov    ax, 40h
          mov    ds, ax
          cli           ;Ints off, critical region.
          mov    bx, TailPtr ;Address where we can put
          push   bx ; next key code.
          mov    [bx], cx ;Store the key code away.
          incptr TailPtr ;Move on to next entry in buf.
          cmp    bx, HeadPtr ;Data overrun?
          jne    StoreOkay ;If not, jump, if so
          pop    TailPtr ; ignore key entry.

```

```

StoreOkay:  sub     sp, 2           ;So stack matches alt path.
             add     sp, 2           ;Remove junk data from stk.
             pop     bx
             pop     ds
             ired                    ;Restores interrupts.

; ExtStatus- Retrieve the extended keyboard status and return it in
;            AH, also returns the standard keyboard status in AL.

ExtStatus:  push    ds
             mov     ax, 40h
             mov     ds, ax

             mov     ah, KbdFlags2
             and     ah, 7Fh         ;Clear final sysreq field.
             test    ah, 100b       ;Test cur sysreq bit.
             je     NoSysReq        ;Skip if it's zero.
             or     ah, 80h         ;Set final sysreq bit.

NoSysReq:   and     ah, 0F0h         ;Clear alt/ctrl bits.
             mov     al, KbdFlags3
             and     al, 1100b       ;Grab rt alt/ctrl bits.
             or     ah, al          ;Merge into AH.
             mov     al, KbdFlags2
             and     al, 11b        ;Grab left alt/ctrl bits.
             or     ah, al          ;Merge into AH.

             mov     al, KbdFlags1   ;AL contains normal flags.
             pop     ds
             ired

; SetAutoRpt- Sets the autorepeat rate. On entry, bh=0, 1, 2, or 3 (delay
;             in 1/4 sec before autorepeat starts) and bl=0..1Fh (repeat
;             rate, about 2:1 to 30:1 (chars:sec).

SetAutoRpt: push    cx
             push   bx

             mov     al, 0ADh       ;Disable kbd for now.
             call   SetCmd

             and     bh, 11b        ;Force into proper range.
             mov     cl, 5
             shl    bh, cl          ;Move to final position.
             and     bl, 1Fh        ;Force into proper range.
             or     bh, bl          ;8042 command data byte.
             mov     al, 0F3h       ;8042 set repeat rate cmd.
             call   SendCmd        ;Send the command to 8042.
             mov     al, bh         ;Get parameter byte
             call   SendCmd        ;Send parameter to the 8042.

             mov     al, 0AEh       ;Reenable keyboard.
             call   SetCmd
             mov     al, 0F4h       ;Restart kbd scanning.
             call   SendCmd

             pop     bx
             pop     cx
             ired

MyInt16     endp

; SetCmd-    Sends the command byte in the AL register to the 8042
;           keyboard microcontroller chip (command register at
;           port 64h).

SetCmd     proc    near
             push   cx
             push   ax             ;Save command value.
             cli                    ;Critical region, no ints now.

```

```

; Wait until the 8042 is done processing the current command.

Wait4Empty:   xor     cx, cx           ;Allow 65,536 times thru loop.
              in     al, 64h       ;Read keyboard status register.
              test   al, 10b       ;Input buffer full?
              loopnz Wait4Empty    ;If so, wait until empty.

; Okay, send the command to the 8042:

              pop     ax           ;Retrieve command.
              out    64h, al
              sti
              pop     cx           ;Okay, ints can happen again.
              ret
SetCmd        endp

; SendCmd- The following routine sends a command or data byte to the
;          keyboard data port (port 60h).

SendCmd       proc     near
              push    ds
              push    bx
              push    cx
              mov     cx, 40h
              mov     ds, cx
              mov     bx, ax       ;Save data byte

RetryLp:      mov     bh, 3        ;Retry cnt.
              cli         ;Disable ints while accessing HW.

; Clear the Error, Acknowledge received, and resend received flags
; in KbdFlags4

              and     byte ptr KbdFlags4, 4fh

; Wait until the 8042 is done processing the current command.

Wait4Empty:   xor     cx, cx           ;Allow 65,536 times thru loop.
              in     al, 64h       ;Read keyboard status register.
              test   al, 10b       ;Input buffer full?
              loopnz Wait4Empty    ;If so, wait until empty.

; Okay, send the data to port 60h

              mov     al, bl
              out    60h, al
              sti         ;Allow interrupts now.

; Wait for the arrival of an acknowledgement from the keyboard ISR:

Wait4Ack:     xor     cx, cx           ;Wait a long time, if need be.
              test   byt KbdFlags4, 10 ;Acknowledge received bit.
              jnz    GotAck
              loop   Wait4Ack
              dec    bh           ;Do a retry on this guy.
              jne   RetryLp

; If the operation failed after 3 retries, set the error bit and quit.

              or     byt KbdFlags4, 80h ;Set error bit.

GotAck:       pop     cx
              pop     bx
              pop     ds
              ret
SendCmd       endp

Main         proc

```

```

        mov     ax, cseg
        mov     ds, ax

        print
        byte   "INT 16h Replacement",cr,lf
        byte   "Installing...",cr,lf,0

; Patch into the INT 9 and INT 16 interrupt vectors. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 and INT 16 values directly into
; the OldInt9 and OldInt16 variables.

        cli                               ;Turn off interrupts!
        mov     ax, 0
        mov     es, ax
        mov     ax, es:[16h*4]
        mov     word ptr OldInt16, ax
        mov     ax, es:[16h*4 + 2]
        mov     word ptr OldInt16+2, ax
        mov     es:[16h*4], offset MyInt16
        mov     es:[16h*4+2], cs
        sti                               ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

        print
        byte   "Installed.",cr,lf,0

        mov     ah, 62h                    ;Get this program's PSP
        int     21h                        ; value.

        mov     dx, EndResident            ;Compute size of program.
        sub     dx, bx
        mov     ax, 3100h                  ;DOS TSR command.
        int     21h

Main
cseg    endp
ends

sseg    segment    para stack 'stack'
stk     db         1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment    para public 'zzzzzz'
LastBytes   db         16 dup (?)
zzzzzzseg    ends
end        Main

```

---

## 20.5 The Keyboard Interrupt Service Routine

The int 16h ISR is the interface between application programs and the keyboard. In a similar vein, the int 9 ISR is the interface between the keyboard hardware and the int 16h ISR. It is the job of the int 9 ISR to process keyboard hardware interrupts, convert incoming scan codes to scan/ASCII code combinations and place them in the typeahead buffer, and process other messages the keyboard generates.

To convert keyboard scan codes to scan/ASCII codes, the int 9 ISR must keep track of the current state of the modifier keys. When a scan code comes along, the int 9 ISR can use the `xlat` instruction to translate the scan code to an ASCII code using a table int 9 selects on the basis of the modifier flags. Another important issue is that the int 9 handler must handle special key sequences like `ctrl-alt-del` (reset) and `PrtSc`. The following assembly code provides a simple int 9 handler for the keyboard. It does not support `alt-Keypad` ASCII code entry or a few other minor features, but it does support almost everything you need for a keyboard interrupt service routine. Certainly it demonstrates all the techniques you need to know when programming the keyboard.

```

; INT9.ASM
;
; A short TSR to provide a driver for the keyboard hardware interrupt.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg          segment    para public 'code'
OldInt9      dword      ?
cseg          ends

; Marker segment, to find the end of the resident section.

EndResident  segment    para public 'Resident'
EndResident  ends

               .xlist
               include  stdlib.a
               includelib stdlib.lib
               .list

NumLockScan  equ         45h
ScrlLockScan equ         46h
CapsLockScan equ         3ah
CtrlScan     equ         1dh
AltScan      equ         38h
RShiftScan   equ         36h
LShiftScan   equ         2ah
InsScanCode  equ         52h
DelScanCode  equ         53h

; Bits for the various modifier keys

RShfBit      equ         1
LShfBit      equ         2
CtrlBit      equ         4
AltBit       equ         8
SLBit        equ         10h
NLBit        equ         20h
CLBit        equ         40h
InsBit       equ         80h

KbdFlags     equ         <byte ptr ds:[17h]>
KbdFlags2    equ         <byte ptr ds:[18h]>
KbdFlags3    equ         <byte ptr ds:[96h]>
KbdFlags4    equ         <byte ptr ds:[97h]>

byp          equ         <byte ptr>

cseg          segment    para public 'code'
               assume    ds:nothing

; Scan code translation table.
; The incoming scan code from the keyboard selects a row.
; The modifier status selects the column.
; The word at the intersection of the two is the scan/ASCII code to
; put into the PC's type ahead buffer.
; If the value fetched from the table is zero, then we do not put the
; character into the type ahead buffer.
;
;           norm  shft  ctrl  alt   num  caps  shcap  shnum
ScanXlat word 0000h, 0000h, 0000h, 0000h, 0000h, 0000h, 0000h, 0000h
               word 011bh, 011bh, 011bh, 011bh, 011bh, 011bh, 011bh, 011bh ;ESC
               word 0231h, 0231h, 0000h, 7800h, 0231h, 0231h, 0231h, 0321h ;! !

```

	word	0332h,	0340h,	0300h,	7900h,	0332h,	0332h,	0332h,	0332h	;2 @
	word	0433h,	0423h,	0000h,	7a00h,	0433h,	0433h,	0423h,	0423h	;3 #
	word	0534h,	0524h,	0000h,	7b00h,	0534h,	0534h,	0524h,	0524h	;4 \$
	word	0635h,	0625h,	0000h,	7c00h,	0635h,	0635h,	0625h,	0625h	;5 %
	word	0736h,	075eh,	071eh,	7d00h,	0736h,	0736h,	075eh,	075eh	;6 ^
	word	0837h,	0826h,	0000h,	7e00h,	0837h,	0837h,	0826h,	0826h	;7 &
	word	0938h,	092ah,	0000h,	7f00h,	0938h,	0938h,	092ah,	092ah	;8 *
	word	0a39h,	0a28h,	0000h,	8000h,	0a39h,	0a39h,	0a28h,	0a28h	;9 (
	word	0b30h,	0b29h,	0000h,	8100h,	0b30h,	0b30h,	0b29h,	0b29h	;0 )
	word	0c2dh,	0c5fh,	0000h,	8200h,	0c2dh,	0c2dh,	0c5fh,	0c5fh	;- _
	word	0d3dh,	0d2bh,	0000h,	8300h,	0d3dh,	0d3dh,	0d2bh,	0d2bh	;= +
	word	0e08h,	0e08h,	0e7fh,	0000h,	0e08h,	0e08h,	0e08h,	0e08h	;bksp
	word	0f09h,	0f00h,	0000h,	0000h,	0f09h,	0f09h,	0f00h,	0f00h	;Tab
;		norm	shft	ctrl	alt	num	caps	shcap	shnum	
	word	1071h,	1051h,	1011h,	1000h,	1071h,	1051h,	1051h,	1071h	;Q
	word	1177h,	1057h,	1017h,	1100h,	1077h,	1057h,	1057h,	1077h	;W
	word	1265h,	1245h,	1205h,	1200h,	1265h,	1245h,	1245h,	1265h	;E
	word	1372h,	1352h,	1312h,	1300h,	1272h,	1252h,	1252h,	1272h	;R
	word	1474h,	1454h,	1414h,	1400h,	1474h,	1454h,	1454h,	1474h	;T
	word	1579h,	1559h,	1519h,	1500h,	1579h,	1559h,	1579h,	1559h	;Y
	word	1675h,	1655h,	1615h,	1600h,	1675h,	1655h,	1675h,	1655h	;U
	word	1769h,	1749h,	1709h,	1700h,	1769h,	1749h,	1769h,	1749h	;I
	word	186fh,	184fh,	180fh,	1800h,	186fh,	184fh,	186fh,	184fh	;O
	word	1970h,	1950h,	1910h,	1900h,	1970h,	1950h,	1970h,	1950h	;P
	word	1a5bh,	1a7bh,	1a1bh,	0000h,	1a5bh,	1a5bh,	1a7bh,	1a7bh	:[ {
	word	1b5dh,	1b7dh,	1b1dh,	0000h,	1b5dh,	1b5dh,	1b7dh,	1b7dh	]; }
	word	1c0dh,	1c0dh,	1c0ah,	0000h,	1c0dh,	1c0dh,	1c0ah,	1c0ah	;enter
	word	1d00h,	1d00h,	1d00h,	1d00h,	1d00h,	1d00h,	1d00h,	1d00h	;ctrl
	word	1e61h,	1e41h,	1e01h,	1e00h,	1e61h,	1e41h,	1e61h,	1e41h	;A
	word	1f73h,	1f5eh,	1f13h,	1f00h,	1f73h,	1f53h,	1f73h,	1f53h	;S
;		norm	shft	ctrl	alt	num	caps	shcap	shnum	
	word	2064h,	2044h,	2004h,	2000h,	2064h,	2044h,	2064h,	2044h	;D
	word	2166h,	2146h,	2106h,	2100h,	2166h,	2146h,	2166h,	2146h	;F
	word	2267h,	2247h,	2207h,	2200h,	2267h,	2247h,	2267h,	2247h	;G
	word	2368h,	2348h,	2308h,	2300h,	2368h,	2348h,	2368h,	2348h	;H
	word	246ah,	244ah,	240ah,	2400h,	246ah,	244ah,	246ah,	244ah	;J
	word	256bh,	254bh,	250bh,	2500h,	256bh,	254bh,	256bh,	254bh	;K
	word	266ch,	264ch,	260ch,	2600h,	266ch,	264ch,	266ch,	264ch	;L
	word	273bh,	273ah,	0000h,	0000h,	273bh,	273bh,	273ah,	273ah	;: :
	word	2827h,	2822h,	0000h,	0000h,	2827h,	2827h,	2822h,	2822h	;‘ “
	word	2960h,	297eh,	0000h,	0000h,	2960h,	2960h,	297eh,	297eh	;` ~
	word	2a00h,	2a00h,	2a00h,	2a00h,	2a00h,	2a00h,	2a00h,	2a00h	;LShf
	word	2b5ch,	2b7ch,	2b1ch,	0000h,	2b5ch,	2b5ch,	2b7ch,	2b7ch	;\
	word	2c7ah,	2c5ah,	2c1ah,	2c00h,	2c7ah,	2c5ah,	2c7ah,	2c5ah	;Z
	word	2d78h,	2d58h,	2d18h,	2d00h,	2d78h,	2d58h,	2d78h,	2d58h	;X
	word	2e63h,	2e43h,	2e03h,	2e00h,	2e63h,	2e43h,	2e63h,	2e43h	;C
	word	2f76h,	2f56h,	2f16h,	2f00h,	2f76h,	2f56h,	2f76h,	2f56h	;V
;		norm	shft	ctrl	alt	num	caps	shcap	shnum	
	word	3062h,	3042h,	3002h,	3000h,	3062h,	3042h,	3062h,	3042h	;B
	word	316eh,	314eh,	310eh,	3100h,	316eh,	314eh,	316eh,	314eh	;N
	word	326dh,	324dh,	320dh,	3200h,	326dh,	324dh,	326dh,	324dh	;M
	word	332ch,	333ch,	0000h,	0000h,	332ch,	332ch,	333ch,	333ch	; , <
	word	342eh,	343eh,	0000h,	0000h,	342eh,	342eh,	343eh,	343eh	; . >
	word	352fh,	353fh,	0000h,	0000h,	352fh,	352fh,	353fh,	353fh	; / ?
	word	3600h,	3600h,	3600h,	3600h,	3600h,	3600h,	3600h,	3600h	;rshf
	word	372ah,	0000h,	3710h,	0000h,	372ah,	372ah,	0000h,	0000h	;* PS
	word	3800h,	3800h,	3800h,	3800h,	3800h,	3800h,	3800h,	3800h	;alt
	word	3920h,	3920h,	3920h,	0000h,	3920h,	3920h,	3920h,	3920h	;spc
	word	3a00h,	3a00h,	3a00h,	3a00h,	3a00h,	3a00h,	3a00h,	3a00h	;caps
	word	3b00h,	5400h,	5e00h,	6800h,	3b00h,	3b00h,	5400h,	5400h	;F1
	word	3c00h,	5500h,	5f00h,	6900h,	3c00h,	3c00h,	5500h,	5500h	;F2
	word	3d00h,	5600h,	6000h,	6a00h,	3d00h,	3d00h,	5600h,	5600h	;F3
	word	3e00h,	5700h,	6100h,	6b00h,	3e00h,	3e00h,	5700h,	5700h	;F4
	word	3f00h,	5800h,	6200h,	6c00h,	3f00h,	3f00h,	5800h,	5800h	;F5
;		norm	shft	ctrl	alt	num	caps	shcap	shnum	
	word	4000h,	5900h,	6300h,	6d00h,	4000h,	4000h,	5900h,	5900h	;F6



```

word 4100h, 5a00h, 6400h, 6e00h, 4100h, 4100h, 5a00h, 5a00h ;F7
word 4200h, 5b00h, 6500h, 6f00h, 4200h, 4200h, 5b00h, 5b00h ;F8
word 4300h, 5c00h, 6600h, 7000h, 4300h, 4300h, 5c00h, 5c00h ;F9
word 4400h, 5d00h, 6700h, 7100h, 4400h, 4400h, 5d00h, 5d00h ;F10
word 4500h, 4500h, 4500h, 4500h, 4500h, 4500h, 4500h, 4500h ;num
word 4600h, 4600h, 4600h, 4600h, 4600h, 4600h, 4600h, 4600h ;scr1
word 4700h, 4737h, 7700h, 0000h, 4737h, 4700h, 4737h, 4700h ;home

word 4800h, 4838h, 0000h, 0000h, 4838h, 4800h, 4838h, 4800h ;up
word 4900h, 4939h, 8400h, 0000h, 4939h, 4900h, 4939h, 4900h ;pgup
word 4a2dh, 4a2dh, 0000h, 0000h, 4a2dh, 4a2dh, 4a2dh, 4a2dh ;-
word 4b00h, 4b34h, 7300h, 0000h, 4b34h, 4b00h, 4b34h, 4b00h ;left
word 4c00h, 4c35h, 0000h, 0000h, 4c35h, 4c00h, 4c35h, 4c00h ;Center
word 4d00h, 4d36h, 7400h, 0000h, 4d36h, 4d00h, 4d36h, 4d00h ;right
word 4e2bh, 4e2bh, 0000h, 0000h, 4e2bh, 4e2bh, 4e2bh, 4e2bh ;+
word 4f00h, 4f31h, 7500h, 0000h, 4f31h, 4f00h, 4f31h, 4f00h ;end

;
norm shft ctrl alt num caps shcap shnum
word 5000h, 5032h, 0000h, 0000h, 5032h, 5000h, 5032h, 5000h ;down
word 5100h, 5133h, 7600h, 0000h, 5133h, 5100h, 5133h, 5100h ;pgdn
word 5200h, 5230h, 0000h, 0000h, 5230h, 5200h, 5230h, 5200h ;ins
word 5300h, 532eh, 0000h, 0000h, 532eh, 5300h, 532eh, 5300h ;del
word 0,0,0,0,0,0,0,0 ; --
word 0,0,0,0,0,0,0,0 ; --
word 0,0,0,0,0,0,0,0 ; --
word 5700h, 0000h, 0000h, 0000h, 5700h, 5700h, 0000h, 0000h ;F11

word 5800h, 0000h, 0000h, 0000h, 5800h, 5800h, 0000h, 0000h ;F12

;*****
;
; AL contains keyboard scan code.

PutInBuffer proc near
push ds
push bx

mov bx, 40h ;Point ES at the BIOS
mov ds, bx ; variables.

; If the current scan code is E0 or E1, we need to take note of this fact
; so that we can properly process cursor keys.

cmp al, 0e0h
jne TryE1
or KbdFlags3, 10b ;Set E0 flag
and KbdFlags3, 0FEh ;Clear E1 flag
jmp Done

TryE1: cmp al, 0e1h
jne DoScan
or KbdFlags3, 1 ;Set E1 flag
and KbdFlags3, 0FDh ;Clear E0 Flag
jmp Done

; Before doing anything else, see if this is Ctrl-Alt-Del:

DoScan: cmp al, DelScanCode
jnz TryIns
mov bl, KbdFlags
and bl, AltBit or CtrlBit ;Alt = bit 3, ctrl = bit 2
cmp bl, AltBit or CtrlBit
jne DoPIB
mov word ptr ds:[72h], 1234h ;Warm boot flag.
jmp dword ptr cs:RebootAdrs ;REBOOT Computer

RebootAdrs dword 0ffff0000h ;Reset address.

; Check for the INS key here. This one needs to toggle the ins bit
; in the keyboard flags variables.

```

```

TryIns:      cmp      al, InsScanCode
             jne      TryInsUp
             or       KbdFlags2, InsBit      ;Note INS is down.
             jmp      doPIB                  ;Pass on INS key.

TryInsUp:    cmp      al, InsScanCode+80h    ;INS up scan code.
             jne      TryLShiftDn
             and     KbdFlags2, not InsBit   ;Note INS is up.
             xor     KbdFlags, InsBit       ;Toggle INS bit.
             jmp      QuitPIB

; Handle the left and right shift keys down here.

TryLShiftDn: cmp      al, LShiftScan
             jne      TryLShiftUp
             or       KbdFlags, LShfBit     ;Note that the left
             jmp      QuitPIB              ; shift key is down.

TryLShiftUp: cmp      al, LShiftScan+80h
             jne      TryRShiftDn
             and     KbdFlags, not LShfBit  ;Note that the left
             jmp      QuitPIB              ; shift key is up.

TryRShiftDn: cmp      al, RShiftScan
             jne      TryRShiftUp
             or       KbdFlags, RShfBit     ;Right shf is down.
             jmp      QuitPIB

TryRShiftUp: cmp      al, RShiftScan+80h
             jne      TryAltDn
             and     KbdFlags, not RShfBit  ;Right shf is up.
             jmp      QuitPIB

; Handle the ALT key down here.

TryAltDn:    cmp      al, AltScan
             jne      TryAltUp
             or       KbdFlags, AltBit     ;Alt key is down.
GotoQPIB:    jmp      QuitPIB

TryAltUp:    cmp      al, AltScan+80h
             jne      TryCtrlDn
             and     KbdFlags, not AltBit   ;Alt key is up.
             jmp      DoPIB

; Deal with the control key down here.

TryCtrlDn:   cmp      al, CtrlScan
             jne      TryCtrlUp
             or       KbdFlags, CtrlBit    ;Ctrl key is down.
             jmp      QuitPIB

TryCtrlUp:   cmp      al, CtrlScan+80h
             jne      TryCapsDn
             and     KbdFlags, not CtrlBit  ;Ctrl key is up.
             jmp      QuitPIB

; Deal with the CapsLock key down here.

TryCapsDn:   cmp      al, CapsLockScan
             jne      TryCapsUp
             or       KbdFlags2, CLBit     ;Capslock is down.
             xor     KbdFlags, CLBit       ;Toggle capslock.
             jmp      QuitPIB

TryCapsUp:   cmp      al, CapsLockScan+80h
             jne      TrySLDn
             and     KbdFlags2, not CLBit   ;Capslock is up.
             call    SetLEDs
             jmp      QuitPIB

```

```

; Deal with the Scroll Lock key down here.

TrySLDn:    cmp     al, ScrlLockScan
            jne     TrySLUp
            or      KbdFlags2, SLBit           ;Scrl lock is down.
            xor     KbdFlags, SLBit          ;Toggle scrl lock.
            jmp     QuitPIB

TrySLUp:    cmp     al, ScrlLockScan+80h
            jne     TryNLDn
            and     KbdFlags2, not SLBit      ;Scrl lock is up.
            call    SetLEDS
            jmp     QuitPIB

; Handle the NumLock key down here.

TryNLDn:    cmp     al, NumLockScan
            jne     TryNLUp
            or      KbdFlags2, NLBit          ;Numlock is down.
            xor     KbdFlags, NLBit          ;Toggle numlock.
            jmp     QuitPIB

TryNLUp:    cmp     al, NumLockScan+80h
            jne     DoPIB
            and     KbdFlags2, not NLBit      ;Numlock is up.
            call    SetLEDS
            jmp     QuitPIB

; Handle all the other keys here:

DoPIB:      test     al, 80h                      ;Ignore other up keys.
            jnz     QuitPIB

; If the H.O. bit is set at this point, we'd best only have a zero in AL.
; Otherwise, this is an up code which we can safely ignore.

            call    Convert
            test    ax, ax                      ;Chk for bad code.
            je      QuitPIB

PutCharInBuf: push   cx
            mov    cx, ax
            mov    ah, 5                      ;Store scan code into
            int    16h                       ; type ahead buffer.
            pop    cx

QuitPIB:    and     KbdFlags3, 0FCh           ;E0, E1 not last code.

Done:       pop    bx
            pop    ds
            ret

PutInBuffer endp

;*****
;
; Convert- AL contains a PC Scan code. Convert it to an ASCII char/Scan
;          code pair and return the result in AX. This code assumes
;          that DS points at the BIOS variable space (40h).

Convert     proc     near
            push    bx

            test   al, 80h                   ;See if up code
            jz     DownScanCode
            mov    ah, al
            mov    al, 0
            jmp    CSDone
            CSDone

```

```

; Okay, we've got a down key. But before going on, let's see if we've
; got an ALT-Keypad sequence.

DownScanCode: mov     bh, 0
               mov     bl, al
               shl     bx, 1           ;Multiply by eight to compute
               shl     bx, 1           ; row index index the scan
               shl     bx, 1           ; code xlat table

; Compute modifier index as follows:
;
;     if alt then modifier = 3

               test    KbdFlags, AltBit
               je      NotAlt
               add     bl, 3
               jmp     DoConvert

;     if ctrl, then modifier = 2

NotAlt:        test    KbdFlags, CtrlBit
               je      NotCtrl
               add     bl, 2
               jmp     DoConvert

; Regardless of the shift setting, we've got to deal with numlock
; and capslock. Numlock is only a concern if the scan code is greater
; than or equal to 47h. Capslock is only a concern if the scan code
; is less than this.

NotCtrl:       cmp     al, 47h
               jb     DoCapsLk
               test   KbdFlags, NLBit           ;Test Numlock bit
               je     NoNumLck
               test   KbdFlags, LShfBit or RShfBit ;Check l/r shift.
               je     NumOnly
               add     bl, 7                   ;Numlock and shift.
               jmp     DoConvert

NumOnly:       add     bl, 4                   ;Numlock only.
               jmp     DoConvert

; If numlock is not active, see if a shift key is:

NoNumLck:     test    KbdFlags, LShfBit or RShfBit ;Check l/r shift.
               je     DoConvert               ;normal if no shift.
               add     bl, 1
               jmp     DoConvert

; If the scan code's value is below 47h, we need to check for capslock.

DoCapsLk:     test    KbdFlags, CLBit           ;Chk capslock bit
               je     DoShift
               test   KbdFlags, LShfBit or RShfBit ;Chk for l/r shift
               je     CapsOnly
               add     bl, 6                   ;Shift and capslock.
               jmp     DoConvert

CapsOnly:     add     bl, 5                   ;Capslock
               jmp     DoConvert

; Well, nothing else is active, check for just a shift key.

DoShift:      test    KbdFlags, LShfBit or RShfBit ;l/r shift.
               je     DoConvert
               add     bl, 1                   ;Shift

DoConvert:    shl     bx, 1                   ;Word array
               mov     ax, ScanXlat[bx]
CSDone:       pop     bx
               ret
Convert:      endp

```

```

; SetCmd-      Sends the command byte in the AL register to the 8042
;              keyboard microcontroller chip (command register at
;              port 64h).

SetCmd        proc      near
              push      cx
              push      ax          ;Save command value.
              cli          ;Critical region, no ints now.

; Wait until the 8042 is done processing the current command.

              xor       cx, cx          ;Allow 65,536 times thru loop.
Wait4Empty:   in        al, 64h        ;Read keyboard status register.
              test      al, 10b        ;Input buffer full?
              loopnz    Wait4Empty     ;If so, wait until empty.

; Okay, send the command to the 8042:

              pop       ax            ;Retrieve command.
              out      64h, al
              sti          ;Okay, ints can happen again.
              pop      cx
              ret
SetCmd        endp

; SendCmd-     The following routine sends a command or data byte to the
;              keyboard data port (port 60h).

SendCmd       proc      near
              push      ds
              push      bx
              push      cx
              mov       cx, 40h
              mov       ds, cx
              mov       bx, ax        ;Save data byte

              mov       bh, 3        ;Retry cnt.
RetryLp:      cli          ;Disable ints while accessing HW.

; Clear the Error, Acknowledge received, and resend received flags
; in KbdFlags4

              and      byte ptr KbdFlags4, 4fh

; Wait until the 8042 is done processing the current command.

              xor       cx, cx          ;Allow 65,536 times thru loop.
Wait4Empty:   in        al, 64h        ;Read keyboard status register.
              test      al, 10b        ;Input buffer full?
              loopnz    Wait4Empty     ;If so, wait until empty.

; Okay, send the data to port 60h

              mov       al, bl
              out      60h, al
              sti          ;Allow interrupts now.

; Wait for the arrival of an acknowledgement from the keyboard ISR:

              xor       cx, cx          ;Wait a long time, if need be.
Wait4Ack:     test      byt KbdFlags4,10h ;Acknowledge received bit.
              jnz      GotAck
              loop     Wait4Ack
              dec      bh            ;Do a retry on this guy.
              jne     RetryLp

; If the operation failed after 3 retries, set the error bit and quit.

              or       byt KbdFlags4,80h ;Set error bit.

```

```

GotAck:      pop      cx
             pop      bx
             pop      ds
             ret
SendCmd     endp

; SetLEDs-   Updates the KbdFlags4 LED bits from the KbdFlags
;           variable and then transmits new flag settings to
;           the keyboard.

SetLEDs     proc      near
             push     ax
             push     cx
             mov      al, KbdFlags
             mov      cl, 4
             shr      al, cl
             and      al, 111b
             and      KbdFlags4, 0F8h   ;Clear LED bits.
             or       KbdFlags4, al     ;Mask in new bits.
             mov      ah, al           ;Save LED bits.

             mov      al, 0ADh         ;Disable kbd for now.
             call     SetCmd

             mov      al, 0EDh         ;8042 set LEDs cmd.
             call     SendCmd          ;Send the command to 8042.
             mov      al, ah          ;Get parameter byte
             call     SendCmd          ;Send parameter to the 8042.

             mov      al, 0AEh         ;Reenable keyboard.
             call     SetCmd
             mov      al, 0F4h         ;Restart kbd scanning.
             call     SendCmd
             pop      cx
             pop      ax
SetLEDs     endp

; MyInt9-    Interrupt service routine for the keyboard hardware
;           interrupt.

MyInt9      proc      far
             push     ds
             push     ax
             push     cx

             mov      ax, 40h
             mov      ds, ax

             mov      al, 0ADh         ;Disable keyboard
             call     SetCmd
             cli                     ;Disable interrupts.

Wait4Data:  xor      cx, cx
             in       al, 64h         ;Read kbd status port.
             test     al, 10b         ;Data in buffer?
             loopz   Wait4Data       ;Wait until data available.
             in       al, 60h         ;Get keyboard data.
             cmp      al, 0EEh        ;Echo response?
             je       QuitInt9
             cmp      al, 0FAh        ;Acknowledge?
             jne     NotAck
             or       KbdFlags4, 10h  ;Set ack bit.
             jmp      QuitInt9

NotAck:     cmp      al, 0FEh         ;Resend command?
             jne     NotResend
             or       KbdFlags4, 20h  ;Set resend bit.
             jmp      QuitInt9

```

; Note: other keyboard controller commands all have their H.O. bit set

```

; and the PutInBuffer routine will ignore them.

NotResend:    call        PutInBuffer        ;Put in type ahead buffer.

QuitInt9:    mov         al, 0AEh          ;Reenable the keyboard
             call        SetCmd

             mov         al, 20h          ;Send EOI (end of interrupt)
             out         20h, al         ; to the 8259A PIC.
             pop         cx
             pop         ax
             pop         ds
             iret

MyInt9       endp

Main         proc
             assume     ds:cseg

             mov         ax, cseg
             mov         ds, ax

             print
             byte       "INT 9 Replacement",cr,lf
             byte       "Installing...",cr,lf,0

; Patch into the INT 9 interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 value directly into
; the OldInt9 variable.

             cli                     ;Turn off interrupts!
             mov         ax, 0
             mov         es, ax
             mov         ax, es:[9*4]
             mov         word ptr OldInt9, ax
             mov         ax, es:[9*4 + 2]
             mov         word ptr OldInt9+2, ax
             mov         es:[9*4], offset MyInt9
             mov         es:[9*4+2], cs
             sti                     ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

             print
             byte       "Installed.",cr,lf,0

             mov         ah, 62h         ;Get this program's PSP
             int         21h           ; value.

             mov         dx, EndResident ;Compute size of program.
             sub         dx, bx
             mov         ax, 3100h      ;DOS TSR command.
             int         21h

Main         endp
cseg         ends

sseg        segment para stack 'stack'
stk         byte       1024 dup ("stack ")
sseg        ends

zzzzzzseg   segment para public 'zzzzzz'
LastBytes   db         16 dup (?)
zzzzzzseg   ends
end         Main

```

## 20.6 Patching into the INT 9 Interrupt Service Routine

For many programs, such as pop-up programs or keyboard enhancers, you may need to intercept certain “hot keys” and pass all remaining scan codes through to the default keyboard interrupt service routine. You can insert an int 9 interrupt service routine into an interrupt nine chain just like any other interrupt. When the keyboard interrupts the system to send a scan code, your interrupt service routine can read the scan code from port 60h and decide whether to process the scan code itself or pass control on to some other int 9 handler. The following program demonstrates this principle; it deactivates the ctrl-alt-del reset function on the keyboard by intercepting and throwing away delete scan codes when the ctrl and alt bits are set in the keyboard flags byte.

```

; NORESET.ASM
;
; A short TSR that patches the int 9 interrupt and intercepts the
; ctrl-alt-del keystroke sequence.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg          segment      para public 'code'
OldInt9      dword        ?
cseg          ends

; Marker segment, to find the end of the resident section.

EndResident  segment      para public 'Resident'
EndResident  ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

DelScanCode  equ          53h

; Bits for the various modifier keys

CtrlBit      equ          4
AltBit       equ          8

KbdFlags     equ          <byte ptr ds:[17h]>

cseg          segment      para public 'code'
                assume     ds:nothing

; SetCmd-      Sends the command byte in the AL register to the 8042
;              keyboard microcontroller chip (command register at
;              port 64h).

SetCmd       proc          near
                push       cx
                push       ax                ;Save command value.
                cli                ;Critical region, no ints now.

; Wait until the 8042 is done processing the current command.

Wait4Empty:  xor          cx, cx                ;Allow 65,536 times thru loop.
                in         al, 64h            ;Read keyboard status register.

```



```

                test     al, 10b           ;Input buffer full?
                loopnz   Wait4Empty       ;If so, wait until empty.

; Okay, send the command to the 8042:

                pop      ax               ;Retrieve command.
                out      64h, al
                sti
                pop      cx               ;Okay, ints can happen again.
                ret
SetCmd         endp

; MyInt9-      Interrupt service routine for the keyboard hardware
;              interrupt. Tests to see if the user has pressed a
;              DEL key. If not, it passes control on to the original
;              int 9 handler. If so, it first checks to see if the
;              alt and ctrl keys are currently down; if not, it passes
;              control to the original handler. Otherwise it eats the
;              scan code and doesn't pass the DEL through.

MyInt9        proc      far
                push     ds
                push     ax
                push     cx

                mov      ax, 40h
                mov      ds, ax

                mov      al, 0ADh        ;Disable keyboard
                call     SetCmd
                cli        ;Disable interrupts.

Wait4Data:    in        cx, cx
                in        al, 64h        ;Read kbd status port.
                test     al, 10b        ;Data in buffer?
                loopz    Wait4Data      ;Wait until data available.

                in        al, 60h        ;Get keyboard data.
                cmp      al, DelScanCode ;Is it the delete key?
                jne      OrigInt9
                mov      al, KbdFlags   ;Okay, we've got DEL, is
                and      al, AltBit or CtrlBit ; ctrl+alt down too?
                cmp      al, AltBit or CtrlBit
                jne      OrigInt9

; If ctrl+alt+DEL is down, just eat the DEL code and don't pass it through.

                mov      al, 0AEh        ;Reenable the keyboard
                call     SetCmd

                mov      al, 20h        ;Send EOI (end of interrupt)
                out      20h, al        ; to the 8259A PIC.
                pop      cx
                pop      ax
                pop      ds
                iret

; If ctrl and alt aren't both down, pass DEL on to the original INT 9
; handler routine.

OrigInt9:     mov      al, 0AEh        ;Reenable the keyboard
                call     SetCmd

                pop      cx
                pop      ax
                pop      ds
                jmp      cs:OldInt9
MyInt9       endp

Main         proc
                assume   ds:cseg

```

```

        mov     ax, cseg
        mov     ds, ax

        print
        byte   "Ctrl-Alt-Del Filter",cr,lf
        byte   "Installing...",cr,lf,0

; Patch into the INT 9 interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 value directly into
; the OldInt9 variable.

        cli                               ;Turn off interrupts!
        mov     ax, 0
        mov     es, ax
        mov     ax, es:[9*4]
        mov     word ptr OldInt9, ax
        mov     ax, es:[9*4 + 2]
        mov     word ptr OldInt9+2, ax
        mov     es:[9*4], offset MyInt9
        mov     es:[9*4+2], cs
        sti                               ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

        print
        byte   "Installed.",cr,lf,0

        mov     ah, 62h                    ;Get this program's PSP
        int     21h                        ; value.

        mov     dx, EndResident            ;Compute size of program.
        sub     dx, bx
        mov     ax, 3100h                  ;DOS TSR command.
        int     21h

Main
cseg    endp
        ends

sseg    segment    para stack 'stack'
stk     db         1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment    para public 'zzzzzz'
LastBytes   db         16 dup (?)
zzzzzzseg    ends
end        Main

```

---

## 20.7 Simulating Keystrokes

At one point or another you may want to write a program that passes keystrokes on to another application. For example, you might want to write a keyboard macro TSR that lets you capture certain keys on the keyboard and send a sequence of keys through to some underlying application. Perhaps you'll want to program an entire string of characters on a normally unused keyboard sequence (e.g., ctrl-up or ctrl-down). In any case, your program will use some technique to pass characters to a foreground application. There are three well-known techniques for doing this: store the scan/ASCII code directly in the keyboard buffer, use the 80x86 *trace* flag to simulate in `al`, 60h instructions, or program the on-board 8042 microcontroller to transmit the scan code for you. The next three sections describe these techniques in detail.

---

### 20.7.1 Stuffing Characters in the Type Ahead Buffer

Perhaps the easiest way to insert keystrokes into an application is to insert them directly into the system's type ahead buffer. Most modern BIOSes provide an `int 16h` function to do this (see "The Keyboard

BIOS Interface” on page 1168). Even if your system does not provide this function, it is easy to write your own code to insert data in the system type ahead buffer; or you can copy the code from the int 16h handler provided earlier in this chapter.

The nice thing about this approach is that you can deal directly with ASCII characters (at least, for those key sequences that are ASCII). You do not have to worry about sending shift up and down codes around the scan code for tn “A” so you can get an upper case “A”, you need only insert 1E41h into the buffer. In fact, most programs ignore the scan code, so you can simply insert 0041h into the buffer and almost any application will accept the funny scan code of zero.

The major drawback to the buffer insertion technique is that many (popular) applications bypass DOS and BIOS when reading the keyboard. Such programs go directly to the keyboard’s port (60h) to read their data. As such, shoving scan/ASCII codes into the type ahead buffer will have no effect. Ideally, you would like to stuff a scan code directly into the keyboard controller chip and have it return that scan code as though someone actually pressed that key. Unfortunately, there is no universally compatible way to do this. However, there are some close approximations, keep reading...

---

## 20.7.2 Using the 80x86 Trace Flag to Simulate IN AL, 60H Instructions

One way to deal with applications that access the keyboard hardware directly is to *simulate* the 80x86 instruction set. For example, suppose we were able to take control of the int 9 interrupt service routine and execute each instruction under our control. We could choose to let all instructions *except* the in instruction execute normally. Upon encountering an in instruction (that the keyboard ISR uses to read the keyboard data), we check to see if it is accessing port 60h. If so, we simply load the al register with the desired scan code rather than actually execute the in instruction. It is also important to check for the out instruction, since the keyboard ISR will want to send an EOI signal to the 8259A PIC after reading the keyboard data, we can simply ignore out instructions that write to port 20h.

The only difficult part is telling the 80x86 to pass control to our routine when encountering certain instructions (like in and out) and to execute other instructions normally. While this is not directly possible in real mode<sup>7</sup>, there is a close approximation we can make. The 80x86 CPUs provide a *trace* flag that generates an exception after the execution of each instruction. Normally, debuggers use the trace flag to single step through a program. However, by writing our own exception handler for the trace exception, we can gain control of the machine between the execution of every instruction. Then, we can look at the opcode of the next instruction to execute. If it is not an in or out instruction, we can simply return and execute the instruction normally. If it is an in or out instruction, we can determine the I/O address and decide whether to simulate or execute the instruction.

In addition to the in and out instructions, we will need to simulate any int instructions we find as well. The reason is because the int instruction pushes the flags on the stack and then clears the trace bit in the flags register. This means that the interrupt service routine associated with that int instruction would execute normally and we would miss any in or out instructions appearing therein. However, it is easy to simulate the int instruction, leaving the trace flag enabled, so we will add int to our list of instructions to interpret.

The only problem with this approach is that it is slow. Although the trace trap routine will only execute a few instructions on each call, it does so for every instruction in the int 9 interrupt service routine. As a result, during simulation, the interrupt service routine will run 10 to 20 times slower than the real code would. This generally isn’t a problem because most keyboard interrupt service routines are very short. However, you might encounter an application that has a large internal int 9 ISR and this method would noticeably slow the program. However, for most applications this technique works just fine and no one will notice any performance loss while they are typing away (slowly) at the keyboard.

---

7. It is possible to trap I/O instructions when running in protected mode.

The following assembly code provides a short example of a trace exception handler that simulates keystrokes in this fashion:

```

        .xlist
        include  stdlib.a
        includelib stdlib.lib
        .list

cseg          segment  para public 'code'
              assume   ds:nothing

; ScanCode must be in the Code segment.

ScanCode     byte      0

;*****
;
; KbdSim- Passes the scan code in AL through the keyboard controller
; using the trace flag. The way this works is to turn on the
; trace bit in the flags register. Each instruction then causes a trace
; trap. The (installed) trace handler then looks at each instruction to
; handle IN, OUT, INT, and other special instructions. Upon encountering
; an IN AL, 60 (or equivalent) this code simulates the instruction and
; returns the specified scan code rather than actually executing the IN
; instruction. Other instructions need special treatment as well. See
; the code for details. This code is pretty good at simulating the hardware,
; but it runs fairly slow and has a few compatibility problems.

KbdSim       proc      near

              pushf
              push     es
              push     ax
              push     bx

              xor      bx, bx           ;Point es at int vector tbl
              mov     es, bx           ; (to simulate INT 9).
              cli     ;No interrupts for now.
              mov     cs:ScanCode, al ;Save output scan code.

              push    es:[1*4]         ;Save current INT 1 vector
              push    es:2[1*4]        ; so we can restore it later.

; Point the INT 1 vector at our INT 1 handler:

              mov     word ptr es:[1*4], offset MyInt1
              mov     word ptr es:[1*4 + 2], cs

; Turn on the trace trap (bit 8 of flags register):

              pushf
              pop     ax
              or      ah, 1
              push   ax
              popf

; Simulate an INT 9 instruction. Note: cannot actually execute INT 9 here
; since INT instructions turn off the trace operation.

              pushf
              call   dword ptr es:[9*4]

```

```

; Turn off the trace operation:

        pushf
        pop     ax
        and     ah, 0feh           ;Clear trace bit.
        push   ax
        popf

; Disable trace operation.

        pop     es:[1*4 + 2]       ;Restore previous INT 1
        pop     es:[1*4]          ; handler.

; Okay, we're done. Restore registers and return.

VMDone:  pop     bx
        pop     ax
        pop     es
        popf
        ret
KbdSim   endp

;-----
;
; MyInt1- Handles the trace trap (INT 1). This code looks at the next
; opcode to determine if it is one of the special opcodes we have to
; handle ourselves.

MyInt1   proc     far
        push   bp
        mov    bp, sp             ;Gain access to return adrs via BP.
        push  bx
        push  ds

; If we get down here, it's because this trace trap is directly due to
; our having punched the trace bit. Let's process the trace trap to
; simulate the 80x86 instruction set.
;
; Get the return address into DS:BX
NextInstr:  lds     bx, 2[bp]

; The following is a special case to quickly eliminate most opcodes and
; speed up this code by a tiny amount.

        cmp    byte ptr [bx], 0cdh ;Most opcodes are less than
        jnb   NotSimple           ; 0cdh, hence we quickly
        pop   ds                  ; return back to the real
        pop   bx                  ; program.
        pop   bp
        iret

NotSimple: je     IsIntInstr       ;If it's an INT instruction.

        mov   bx, [bx]           ;Get current instruction's opcode.
        cmp  bl, 0e8h           ;CALL opcode
        je   ExecInstr
        jb   TryInOut0

        cmp  bl, 0ech           ;IN al, dx instr.
        je  MaybeIn60
        cmp  bl, 0eeh           ;OUT dx, al instr.
        je  MaybeOut20
        pop  ds                 ;A normal instruction if we get
        pop  bx                 ; down here.
        pop  bp
        iret

```

```

TryInOut0:    cmp     bx, 60e4h    ;IN al, 60h instr.
              je      IsINAL60
              cmp     bx, 20e6h    ;out 20, al instr.
              je      IsOut20

; If it wasn't one of our magic instructions, execute it and continue.

ExecInstr:    pop     ds
              pop     bx
              pop     bp
              iret

; If this instruction is IN AL, DX we have to look at the value in DX to
; determine if it's really an IN AL, 60h instruction.

MaybeIn60:   cmp     dx, 60h
              jne     ExecInstr
              inc     word ptr 2[bp] ;Skip over this 1 byte instr.
              mov     al, cs:ScanCode
              jmp     NextInstr

; If this is an IN AL, 60h instruction, simulate it by loading the current
; scan code into AL.

IsInAL60:     mov     al, cs:ScanCode
              add     word ptr 2[bp], 2 ;Skip over this 2-byte instr.
              jmp     NextInstr

; If this instruction is OUT DX, AL we have to look at DX to see if we're
; outputting to location 20h (8259).

MaybeOut20:  cmp     dx, 20h
              jne     ExecInstr
              inc     word ptr 2[bp] ;Skip this 1 byte instruction.
              jmp     NextInstr

; If this is an OUT 20h, al instruction, simply skip over it.

IsOut20:      add     word ptr 2[bp], 2 ;Skip instruction.
              jmp     NextInstr

; IsIntInstr- Execute this code if it's an INT instruction.
;
; The problem with the INT instructions is that they reset the trace bit
; upon execution. For certain guys (see above) we can't have that.
;
; Note: at this point the stack looks like the following:
;
;     flags
;
;     rtn cs --
;           |
;     rtn ip  +-- Points at next instr the CPU will execute.
;
;     bp
;     bx
;     ds
;
; We need to simulate the appropriate INT instruction by:
;
;     (1) adding two to the return address on the stack (so it returns
;         beyond the INT instruction.
;     (2) pushing the flags onto the stack.
;     (3) pushing a phony return address onto the stack which simulates
;         the INT 1 interrupt return address but which "returns" us to
;         the specified interrupt vector handler.
;
; All this results in a stack which looks like the following:
;
;     flags
;
;     rtn cs --

```

```

;
;   rtn ip  +-- Points at next instr beyond the INT instruction.
;
;   flags   --- Bogus flags to simulate those pushed by INT instr.
;
;   rtn cs  +-
;
;   rtn ip  +-- "Return address" which points at the ISR for this INT.
;   bp
;   bx
;   ds

IsINTInstr:  add     word ptr 2[bp], 2 ;Bump rtn adrs beyond INT instr.
            mov     bl, 1[bx]
            mov     bh, 0
            shl     bx, 1           ;Multiply by 4 to get vector
            shl     bx, 1           ; address.

            push    [bp-0]         ;Get and save BP
            push    [bp-2]         ;Get and save BX.
            push    [bp-4]         ;Get and save DS.

            push    cx
            xor     cx, cx         ;Point DS at interrupt
            mov     ds, cx         ; vector table.

            mov     cx, [bp+6]     ;Get original flags.
            mov     [bp-0], cx     ;Save as pushed flags.

            mov     cx, ds:2[bx]   ;Get vector and use it as
            mov     [bp-2], cx     ; the return address.
            mov     cx, ds:[bx]
            mov     [bp-4], cx

            pop     cx
            pop     ds
            pop     bx
            pop     bp
            iret

;
MyInt1      endp

```

```

; Main program - Simulates some keystrokes to demo the above code.

```

```

Main        proc

            mov     ax, cseg
            mov     ds, ax

            print   "Simulating keystrokes via Trace Flag", cr, lf
            byte   "This program places 'DIR' in the keyboard buffer"
            byte   cr, lf, 0

            mov     al, 20h        ;"D" down scan code
            call    KbdSim
            mov     al, 0a0h       ;"D" up scan code
            call    KbdSim

            mov     al, 17h       ;"I" down scan code
            call    KbdSim
            mov     al, 97h       ;"I" up scan code
            call    KbdSim

            mov     al, 13h       ;"R" down scan code
            call    KbdSim
            mov     al, 93h       ;"R" up scan code
            call    KbdSim

            mov     al, 1Ch       ;Enter down scan code

```

```

                                call    KbdSim
                                mov     al, 9Ch           ;Enter up scan code
                                call    KbdSim

                                ExitPgm
Main                             endp

cseg                              ends

sseg          segment      para stack 'stack'
stk           byte        1024 dup ("stack ")
sseg          ends

zzzzzzseg    segment      para public 'zzzzzz'
LastBytes    db           16 dup (?)
zzzzzzseg    ends
end           Main

```

---

### 20.7.3 Using the 8042 Microcontroller to Simulate Keystrokes

Although the trace flag based “keyboard stuffer” routine works with most software that talks to the hardware directly, it still has a few problems. Specifically, it doesn’t work at all with programs that operate in protected mode via a “DOS Extender” library (programming libraries that let programmers access more than one megabyte of memory while running under DOS). The last technique we will look at is to program the on-board 8042 keyboard microcontroller to transmit a keystroke for us. There are two ways to do this: the PS/2 way and the hard way.

The PS/2’s microcontroller includes a command specifically designed to return user programmable scan codes to the system. By writing a 0D2h byte to the controller command port (64h) and a scan code byte to port 60h, you can force the controller to return that scan code as though the user pressed a key on the keyboard. See “The Keyboard Hardware Interface” on page 1159 for more details.

Using this technique provides the most compatible (with existing software) way to return scan codes to an application. Unfortunately, this trick only works on machines that have keyboard controllers that are compatible with the PS/2’s; this is not the majority of machines out there. However, if you are writing code for PS/2s or compatibles, this is the best way to go.

The keyboard controller on the PC/AT and most other PC compatible machines does not support the 0D2h command. Nevertheless, there is a sneaky way to force the keyboard controller to transmit a scan code, if you’re willing to break a few rules. This trick may not work on all machines (indeed, there are many machines on which this trick is known to fail), but it does provide a workaround on a large number of PC compatible machines.

The trick is simple. Although the PC’s keyboard controller doesn’t have a command to return a byte you send it, it does provide a command to return the keyboard controller command byte (KCCB). It also provides another command to write a value to the KCCB. So by writing a value to the KCCB and then issuing the read KCCB command, we can trick the system into returning a user programmable code. Unfortunately, the KCCB contains some undefined reserved bits that have different meanings on different brands of keyboard microcontroller chips. That is the main reason this technique doesn’t work with all machines. The following assembly code demonstrates how to use the PS/2 and PC keyboard controller stuffing methods:

```

                                .xlist
                                include  stdlib.a
                                includelib stdlib.lib
                                .list

cseg          segment      para public 'code'

```



```

                assume     ds:nothing

;*****
;
; PutInATBuffer-
;
; The following code sticks the scan code into the AT-class keyboard
; microcontroller chip and asks it to send the scan code back to us
; (through the hardware port).
;
; The AT keyboard controller:
;
; Data port is at I/O address 60h
; Status port is at I/O address 64h (read only)
; Command port is at I/O address 64h (write only)
;
; The controller responds to the following values sent to the command port:
;
; 20h - Read Keyboard Controller's Command Byte (KCCB) and send the data to
; the data port (I/O address 60h).
;
; 60h - Write KCCB. The next byte written to I/O address 60h is placed in
; the KCCB. The bits of the KCCB are defined as follows:
;
;         bit 7- Reserved, should be a zero
;         bit 6- IBM industrial computer mode.
;         bit 5- IBM industrial computer mode.
;         bit 4- Disable keyboard.
;         bit 3- Inhibit override.
;         bit 2- System flag
;         bit 1- Reserved, should be a zero.
;         bit 0- Enable output buffer full interrupt.
;
;         AAh - Self test
;         ABh - Interface test
;         ACh - Diagnostic dump
;         ADh - Disable keyboard
;         AEh - Enable keyboard
;         C0h - Read Keyboard Controller input port (equip installed)
;         D0h - Read Keyboard Controller output port
;         D1h - Write Keyboard Controller output port
;         E0h - Read test inputs
;         F0h - FFh - Pulse Output port.
;
; The keyboard controller output port is defined as follows:
;
;         bit 7 - Keyboard data (output)
;         bit 6 - Keyboard clock (output)
;         bit 5 - Input buffer empty
;         bit 4 - Output buffer full
;         bit 3 - undefined
;         bit 2 - undefined
;         bit 1 - Gate A20
;         bit 0 - System reset (0=reset)
;
; The keyboard controller input port is defined as follows:
;
;         bit 7 - Keyboard inhibit switch (0=inhibited)
;         bit 6 - Display switch (0=color, 1= mono)
;         bit 5 - Manufacturing jumper
;         bit 4 - System board RAM (0=disable 2nd 256K RAM on system board).
;         bits 0-3 - undefined.
;
; The keyboard controller status port (64h) is defined as follows:
;
;         bit 1 - Set if input data (60h) not available.
;         bit 0 - Set if output port (60h) cannot accept data.

PutInATBuffer proc     near
                assume     ds:nothing
                pushf
                push      ax

```

```

        push    bx
        push    cx
        push    dx

        mov     dl, al           ;Save char to output.

; Wait until the keyboard controller does not contain data before
; proceeding with shoving stuff down its throat.

WaitWhlFull:  xor     cx, cx
              in     al, 64h
              test   al, 1
              loopnz WaitWhlFull

; First things first, let's mask the interrupt controller chip (8259) to
; tell it to ignore interrupts coming from the keyboard. However, turn the
; interrupts on so we properly process interrupts from other sources (this
; is especially important because we're going to wind up sending a false
; EOI to the interrupt controller inside the INT 9 BIOS routine).

        cli
        in     al, 21h         ;Get current mask
        push   ax             ;Save intr mask
        or     al, 2          ;Mask keyboard interrupt
        out    21h, al

; Transmit the desired scan code to the keyboard controller. Call this
; byte the new keyboard controller command (we've turned off the keyboard,
; so this won't affect anything).
;
; The following code tells the keyboard controller to take the next byte
; sent to it and use this byte as the KCCB:

        call   WaitToXmit
        mov    al, 60h        ;Write new KCCB command.
        out    64h, al

; Send the scan code as the new KCCB:

        call   WaitToXmit
        mov    al, dl
        out    60h, al

; The following code instructs the system to transmit the KCCB (i.e., the
; scan code) to the system:

        call   WaitToXmit
        mov    al, 20h        ;"Send KCCB" command.
        out    64h, al

Wait4OutFull: xor    cx, cx
              in     al, 64h
              test   al, 1
              loopz  Wait4OutFull

; Okay, Send a 45h back as the new KCCB to allow the normal keyboard to work
; properly.

        call   WaitToXmit
        mov    al, 60h
        out    64h, al

        call   WaitToXmit
        mov    al, 45h
        out    60h, al

; Okay, execute an INT 9 routine so the BIOS (or whoever) can read the key
; we just stuffed into the keyboard controller. Since we've masked INT 9
; at the interrupt controller, there will be no interrupt coming along from
; the key we shoved in the buffer.

```

```

DoInt9:      in          al, 60h          ;Prevents ints from some codes.
             int         9              ;Simulate hardware kbd int.

; Just to be safe, reenable the keyboard:

             call        WaitToXmit
             mov         al, 0aeh
             out         64h, al

; Okay, restore the interrupt mask for the keyboard in the 8259a.

             pop         ax
             out         21h, al

             pop         dx
             pop         cx
             pop         bx
             pop         ax
             popf
             ret

PutInATBuffer endp

; WaitToXmit- Wait until it's okay to send a command byte to the keyboard
; controller port.

WaitToXmit  proc         near
             push        cx
             push        ax
TstCmdPortLp: in         al, 64h
             test        al, 2          ;Check cntrlr input buffer full flag.
             loopnz     TstCmdPortLp
             pop         ax
             pop         cx
             ret

WaitToXmit  endp

;*****
;
; PutInPS2Buffer- Like PutInATBuffer, it uses the keyboard controller chip
; to return the keycode. However, PS/2 compatible controllers
; have an actual command to return keycodes.

PutInPS2Buffer proc     near
             pushf
             push        ax
             push        bx
             push        cx
             push        dx

             mov         dl, al        ;Save char to output.

; Wait until the keyboard controller does not contain data before
; proceeding with shoving stuff down its throat.

             xor         cx, cx
WaitWhlFull: in         al, 64h
             test        al, 1
             loopnz     WaitWhlFull

; The following code tells the keyboard controller to take the next byte
; sent to it and return it as a scan code.

             call        WaitToXmit
             mov         al, 0d2h      ;Return scan code command.
             out         64h, al

```

```

; Send the scan code:

        call    WaitToXmit
        mov     al, dl
        out    60h, al

        pop    dx
        pop    cx
        pop    bx
        pop    ax
        popf
        ret
PutInPS2Buffer endp

; Main program - Simulates some keystrokes to demo the above code.

Main    proc

        mov     ax, cseg
        mov     ds, ax

        print
        byte    "Simulating keystrokes via Trace Flag", cr, lf
        byte    "This program places 'DIR' in the keyboard buffer"
        byte    cr, lf, 0

        mov     al, 20h           ;"D" down scan code
        call    PutInATBuffer
        mov     al, 0a0h         ;"D" up scan code
        call    PutInATBuffer

        mov     al, 17h         ;"I" down scan code
        call    PutInATBuffer
        mov     al, 97h         ;"I" up scan code
        call    PutInATBuffer

        mov     al, 13h         ;"R" down scan code
        call    PutInATBuffer
        mov     al, 93h         ;"R" up scan code
        call    PutInATBuffer

        mov     al, 1Ch         ;Enter down scan code
        call    PutInATBuffer
        mov     al, 9Ch         ;Enter up scan code
        call    PutInATBuffer

        ExitPgm
Main    endp

cseg    ends

sseg    segment    para stack 'stack'
stk     byte    1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment    para public 'zzzzzz'
LastBytes    db    16 dup (?)
zzzzzzseg    ends
end        Main

```

---

## 20.8 Summary

This chapter might seem excessively long for such a mundane topic as keyboard I/O. After all, the Standard Library provides only one primitive routine for keyboard input, `getc`. However, the keyboard on the PC is a complex beast, having no less than two specialized microprocessors controlling it. These microprocessors accept commands from the PC and send commands and data to the PC. If you want to

write some tricky keyboard handling code, you need to have a firm understanding of the keyboard's underlying hardware.

This chapter began by describing the actions the system takes when a user presses a key. As it turns out, the system transmits two *scan codes* every time you press a key – one scan code when you press the key and one scan code when you release the key. These are called down codes and up codes, accordingly. The scan codes the keyboard transmits to the system have little relationship to the standard ASCII character set. Instead, the keyboard uses its own character set and relies upon the keyboard interrupt service routine to translate these scan codes to their appropriate ASCII codes. Some keys do not have ASCII codes, for these keys the system passes along an *extended key code* to the application requesting keyboard input. While translating scan codes to ASCII codes, the keyboard interrupt service routine makes use of certain BIOS flags that track the position of the *modifier* keys. These keys include the shift, ctrl, alt, capslock, and numlock keys. These keys are known as modifiers because they modify the normal code produced by keys on the keyboard. The keyboard interrupt service routine stuffs incoming characters in the system *type ahead buffer* and updates other BIOS variables in segment 40h. An application program or other system service can access this data prepared by the keyboard interrupt service routine. For more information, see

- “Keyboard Basics” on page 1153

The PC interfaces to the keyboard using two separate microcontroller chips. These chips provide user programming registers and a very flexible command set. If you want to program the keyboard beyond simply reading the keystrokes produced by the keyboard (i.e., manipulate the LEDs on the keyboard), you will need to become familiar with the registers and command sets of these microcontrollers. The discussion of these topics appears in

- “The Keyboard Hardware Interface” on page 1159

Both DOS and BIOS provide facilities to read a key from the system's type ahead buffer. As usual, BIOS' functions provide the most flexibility in terms of getting at the hardware. Furthermore, the BIOS int 16h routine lets you check shift key status, stuff scan/ASCII codes into the type ahead buffer, adjust the autorepeat rate, and more. Given this flexibility, it is difficult to understand why someone would want to talk directly to the keyboard hardware, especially considering the compatibility problems that seem to plague such projects. To learn the proper way to read characters from the keyboard, and more, see

- “The Keyboard DOS Interface” on page 1167
- “The Keyboard BIOS Interface” on page 1168

Although accessing the keyboard hardware directly is a bad idea for most applications, there is a small class of programs, like keyboard enhancers and pop-up programs, that really do need to access the keyboard hardware directly. These programs must supply an interrupt service routine for the int 9 (keyboard) interrupt. For all the details, see:

- “The Keyboard Interrupt Service Routine” on page 1174
- “Patching into the INT 9 Interrupt Service Routine” on page 1184

A keyboard macro program (keyboard enhancer) is a perfect example of a program that might need to talk directly to the keyboard hardware. One problem with such programs is that they need to pass characters along to some underlying application. Given the nature of applications present in the world, this can be a difficult task if you want to be compatible with a large number of PC applications. The problems, and some solutions, appear in

- “Simulating Keystrokes” on page 1186
- “Stuffing Characters in the Type Ahead Buffer” on page 1186
- “Using the 80x86 Trace Flag to Simulate IN AL, 60H Instructions” on page 1187
- “Using the 8042 Microcontroller to Simulate Keystrokes” on page 1192

