DEPARTMENTS

FIRMWARE FURNACE

Ed Nisley

Journey to the Protected Land: Of Characters and Keystrokes

Continuing last month's keyboard discussion, Ed sets out to convert Scan Code Set 3 codes into those expected by the real-mode BIOS. He also looks at how the numerous shift keys are handled.

As you saw last month, Scan Code Set 3 reduces the keyboard interface to the falling-off-a-log level. If your brain works like mine, though, you'd rather not learn a whole new set of keyboard scan codes. The solution is, naturally enough, a simple matter of firmware.

This month, we'll convert Set 3 scan codes into the system scan codes used by the real-mode BIOS and, in the bargain, learn a lot about handling those pesky shift keys. Even though the FFTS keyboard interface doesn't precisely mimic all of the BIOS's peculiarities, the end result is familiar enough.

You can do this trick in real mode, too, but that's a whole 'nother subject!

FIELDING THE INTERRUPT

It's probably worth reviewing how we got here.

After you reset the PC, the BIOS performs its usual power-on testing and puts the keyboard into the default Scan Code Set 2 mode. Our PMLoader program disables all interrupts before switching to 32-bit protected mode and passing control to the FFTS setup code.

The code last month tested the system keyboard controller and the keyboard, then sent the commands and data required to activate Scan

Listing 1—*This interrupt handler responds to each IRQ 1 from the system keyboard controller. Pressing a key sends its single-byte make code to the controller and releasing it sends an F0 byte followed by the key's make code. A simple ring buffer holds the make and break codes until the main keyboard routine can process them.*

```
     UDATASEG
BreakCode   DD ?                    ; holds FO 00 when break found

     CODESEG

     PROC   KeyHandler
     USES   EAX,EDX,DS              ; we can't use automatic restores

     MOV    EAX,GDT_DATA            ; get addressability to our data
     MOV    DS,AX

     IN     AL,KEY_DATA             ; read scan code from controller
     MOVZX  EAX,AL                  ; clear high bytes
     Punt

     CMP    AL,0F0h                 ; are we getting a break code?
     JNE    @@Make
     XCHG   AH,AL                   ; yes, set up F0 00
     MOV    [BreakCode],EAX         ; . . . for next time
     JMP    @Done                   ; and bail out

@Make:
     CMP    [SRCount],RING_SIZE; room for one more?
     JB     @@Insert                ; yes, tuck it away
     MOV    [BreakCode],0           ; no, flush break
     JMP    @@Done                  ; and discard it

@Insert:
     MOV    EDX,[SRHead]            ; aim at ring head
     OR     EAX,[BreakCode]         ; combine with break
     MOV    [BreakCode],0           ; ... clear reminder
     MOV    [ScanRing + EDX*4],EAX; save the code
     INC    [SRCount]               ; account for it
     INC    EDX                     ; tick and wrap index
     CMP    EDX,RING_SIZE
     JB     @@NoWrap
     XOR    EDX,EDX
@@NoWrap:
     MOV    [SRHead],EDX

@@Done:
     MOV    AL,NS_EOI               ; now reset the 8259 ISR
     OUT    I8259A,AL               ; EOI to primary controller

     POP    DS                      ;IRET bypasses the automatic pops
     POP    EDX
     POP    EAX

     IRET                           ; return to interrupted code

     ENDP   KeyHandler
```

Code Set 3. Some of the keyboards I tested had different Set 3 implementations, differing mostly in which keys were typematic and which were make-break. The set-up code reprogrammed the keys in the hope that all the keyboards would then work alike.

The set-up code aimed IRQ 1 at the 32-bit PM interrupt handler shown in Listing 1. The keyboard controller activates IRQ 1 when it has received and processed a byte from the keyboard. All we need do is read a single input port and process the code.

You'll recall that Scan Code Set 3 produces a single-byte make code when each key is active. Typematic keys repeat that make code at a fixed rate. Both typematic and make-break keys produce a two-byte break code when they're released: FO followed by the key's make code. Make-only keys, as you might expect, do not produce a break code.

When the keyboard is in Scan Code Set 2, the codes produced for each key depend on the state of the shift keys. Operation in Scan Code Set 3, on the other hand, produces a unique code for each key regardless of the shift state. It's up to the FFTS keyboard handler to figure out how the shifts affect the key.

The IRQ 1 handler in Listing 1 places scan codes from the keyboard controller into a ring buffer. When it reads an FO code from the controller, it sets a flag indicating that a break code is in progress and exits without changing the buffer.

When the next scan code arrives, the handler adds an FO flag to the high byte and tucks it into the ring buffer. In effect, the ring entries are just a "parallelized" version of the one- or two-byte keyboard codes.

The keyboard always sends both bytes of a break code in quick succession and, if left alone, never interposes anything else between the two. The Official IBM Enhanced Keyboard doc says that you may send commands to the keyboard at any time. That certainly implies that you may interrupt a break sequence with a command.

The real-mode BIOS tracks the keyboard state, presumably to avoid stepping on lengthy scan code sequences. The BIOS keyboard data word at 0040:0096 has several interesting flags that you should ponder before doing any real-mode surgery, particularly in Scan Code Set 2.

I simply ignored the problem, as I couldn't think of a good way to test the ensuing code. You may want to draw some state and timing diagrams when you build your interface. My guess is that many keyboards will do the right thing and, as always, others will fail at the most inopportune time. Program defensively!

There is another trap lying in wait for you assembly-language folks. The USES EAX, EDX, DS directive creates a

Listing 2—*Tasks call this routine to retrieve keystrokes from the ring buffer. The* **Key** *DoSpecials and KeyMakeChar routines translate raw Scan Code Set 3 values into their real-mode BIOS equivalents.*

```
        CODESEG

        PROC    KeyGtGetKey FAR
        USES    EDX,DS,FS

        MOV     EAX,GDT_DATA
        MOV     DS,AX

        MOV     EAX,GDT_CONST
        MOV     FS,AX

        MOV     EAX,[KeyEnable]   ; are we enabled?
        CMP     EAX,0
        JE      @Done             ; nope. return nothing

;--- extract a key or die trying
@@NextKey:
        MOV     EAX,[SRCount]     ; fetch current counter
        CMP     EAX,0             ; if zero,
        JE      @@Done            ; we are done!

        MOV     EDX,[SRTail]      aim at oldest entry
        MOV     EAX,[ScanRing  +  EDX*4];... fetch it

        DEC     [SRCount]         ; account for it
        INC     EDX               ; tick and wrap index
        CMP     EDX,RING_SIZE
        JB      @@NoWrap
        XOR     EDX,EDX
@@NoWrap:
        MOV     [SRTail],EDX

;--- process special keys
        CALL    KeyDoSpecials     ; process special keys
        CMP     EAX,0             ; anything to return?
        JE      @@NextKey         ; no, so get another one

 --- process shift states
        CALL    KeyMakeChar       ; convert into a character
        CMP     EAX,0             ; anything to return?
        JE      @@NextKey         ; no, so get another one

@@Done:
        RET

        ENDP    KeyGtGetKey
```

few lines of code to push those registers onto the stack. In normal routines ending with a RET instruction, the assembler generates a short epilog that restores the registers before the actual RET. This is quite convenient because you only need to enter the registers in one spot and you won't get mismatched pushes and pops.

Interrupt handlers, however, must end with an I RET. That small difference throws the assembler off track: it doesn't produce the epilog. You must manually restore the registers saved by the US ES directive. Nope, this isn't documented anywhere I could find. Nothing is ever simple, is it?

READING CODES

Application programs (if I may so glorify the taskettes) extract keystrokes from the buffer through the KeyGtGet Key call gate. Because FFTS is a cooperative multitasking operating system (well, sort of), KeyGtGetKey returns either a character code or a binary zero when a key isn't available.

Listing 3—*The FFTS keyboard interface returns values compatible with real-mode BIOS key scan codes and characters in AH and AL. This record shows those two bytes along with the 16 shift state bits that fill the remainder of the 32-bit EAX register.*

```
RECORD KEY-SHIFTS {
KEY_SH_SYSREQ_DN:1,      ; 15 Sys Req is pressed
KEY_SH_CAPSLOCK_DN:1,    ; 14 Caps Lock is pressed
KEY_SH_NUMLOCK_DN:1,     ; 13 Num Lock is pressed
KEY_SH_SCROLLLOCK_DN:1, ;12 Scroll Lock is pressed

KEY_SH_ALTRIGHT_DN:1,    ;11 Right Alt is presssed
KEY_SH_CTRLRIGHT_DN:1,;10 Right Ctrl is pressed
KEY_SH_ALTLEFT_DN:1,     ; 9 Left Alt is pressed
KEY_SH_CTRLLEFT_DN:1,    ; 8 Left Ctrl is pressed

KEY_SH_INSERT:1,         ; 7 Insert mode
KEY_SH_CAPSLOCK:1,       ;6 Caps Lock
KEY_SH_NUMLOCK:1,        ; 5 Num Lock
KEY_SH_SCROLLLOCK:1,     ; 4 Scroll Lock

KEY_SH_ALT_DN:1,         ; 3 either Alt key is pressed
KEY_SH_CTRL_DN:1,        ; 2 either Ctrl key is pressed
KEY_SH_SHIFTLEFT_DN:1,   ;  1 Left Shift key is pressed
KEY_SH_SHIFTRIGHT_DN:1,  ;  0 Right Shift key is pressed

KEY_SCANCODE:8,          ; key scan code
KEY_CHARACTER:8          ; processed character
}
```

Listing *4-The three keyboard LEDs must be updated when any one of them changes. This routine maps the* shift *state* info *the LED bit locations and* then *sends the byfe to the keyboard. The command sequence is fair/y lengthy, so the code sends the new bits on/y if they're different from the previous value stored in LastLEDs.*

```
     PROC KeyUpdateLEDs
     USES EAX,EBX

     XOR   EBX,EBX
     TEST  [ShiftState],MASK KEY_SH_CAPSLOCK
     JZ    @@NoCaps
     OR    BL,LED_CAPS
@@NoCaps:
     TEST  [ShiftState],MASK KEY_SH_NUMLOCK
     JZ    @@NoNum
     OR    BL,LED_NUM
@@NoNum:
     TEST [ShiftState],MASK  KEY_SH_SCROLLLOCK
     JZ    @@NoScroll
     OR    BL,LED_SCROLL
@@NoScroll:
     CMP   EBX,[LastLEDs]      ; any change?
     JE    @Done

     MOV   [LastLEDs],EBX      ; yes, save new state

     CALL  KeySendCmdData,CCMD_WRCMD,CMD_NOINTS

     CALL  KeySendDataAck,KCMD_WRLEDS
     CALL  KeySendDataAck,EBX

     CALL  KeySendCmdData,CCMD_WRCMD,CMD_NORMAL

@@Done:
     RET
     ENDP KeyUpdateLEDs
```

a)

Name	DE-25 oin	DE-9 pin	Board Header	DE-9 pin	DB-25 pin	Name
CD	8	1	1 2	6	6	DSR
RD	3	2	3 4	7	4	RTS
TD	2	3	5 6	8	5	CTS
DTR	20	4	7 8	9	22	RI
Gnd	7	5	9 10	n/c	n/c	n/c

b)

Name	DB-25 pin	DE-9 pin	Board Header	DE-9 pin	DB-25 pin	Name
CD	8		1 2	2	3	RD
TD	2	3	3 4	4	20	DTR
Gnd	7	5	5 6	6	6	DSR
RTS	4	7	7 8	8	5	CTS
RI	22	9	9 10	n/c	n/c	n/c

Table *I-The PC's serial port connector pinouts are standardized. Af the other end of the ribbon cable, however, there are two different 2 x 5 pin header configurations. This figure gives a fop view of the headers and the corresponding connector pins. The difference becomes obvious when you compare the two DE-9 layouts.*

It does not stall while waiting for a keystroke, lest the whole system stall with it.

KeyGtGetKey, shown in Listing 2, extracts the next entry from the ring buffer and sends it to two routines for further processing. KeyDoSpecials handles shift, lock, and other oddball keys. KeyMakeChar translates the other keys from Scan Code Set 3 into their real-mode BIOS equivalents.

As a result, KeyGtGetKey returns key scan codes that have nothing whatsoever to do with Scan Code Set 3. Even though I know this, even though I wrote the code, I still stumble while looking at the screen because I expect Scan Code Set 3 values. The upside, once you get used to it, is that all your standard PC reference books remain applicable: there's nothing new to learn about characters and scan codes.

KeyGtGetKey does not precisely mimic the real-mode BIOS response to each and every possible keystroke sequence. In particular, key combinations such as Ctrl-Break, Pause, Shift-PrtSc, Alt-SysReq, and Ctrl-Alt-Del don't behave as you'd expect. Adding some features is just a simple matter of software, others are impractical, and still others are political. As an example of the latter, I decided early on that Ctrl-Alt-Del just wasn't going to reboot the system!

That cavalier attitude would be catastrophic in a commercial operating system running standard DOS apps. There is no PC feature so insignificant that some program won't misbehave if the program is not exactly right. The PC Compatibility Barnacles are very, very solid.

FFTS is an entirely different kettle of fish. In fact, there is no reason why KeyGtGetKey should return real-mode BIOS values instead of sushi shift bits and simmered scan codes. I figured it would be easier to use it this way, but you and your application may have different requirements.

Fortunately, we have control over every program that will ever run with FFTS and can easily adapt to any self-inflicted peculiarities. At worst, we can rewrite the code to make the answer come out right.

The details lie in two routines that process each keystroke: KeyDoSpecialsandKeyMakeChar. If you need some changes, that's where you begin twiddling. Let's begin by examining what must be done.

SPECIAL ORDERS

When you press the Q key, you pretty much know what should happen: a "Q" should appear on the screen. It's not quite that intuitive, however, because you'll actually see a lowercase "q" unless you also press a shift key at the same time. If you do any typing at all, that distinction is buried in your muscle memory and you may have trouble remembering the two keystrokes separately.

The Ctrl, Shift, and Alt keys are collectively known as shift keys because they modify the result of a key stroke. The Enhanced keyboard has two of each shift key, although we expect the same result from either one of the pair. All shift keys work the same way: you must hold them down while pressing another key.

The keyboard also sports three locking shift keys: Caps Lock, Num Lock, and Scroll Lock. Because the actions associated with these keys toggle on and off, you don't have to hold them down. Each has a keyboard LED. But, contrary to some PC mythology, the LEDs aren't linked directly to the keys. We'll see how to drive the LEDs later.

The Insert key behaves like a locking shift key, although it doesn't affect the characters returned by the keyboard interface. Instead, the program using the keystrokes either replaces existing text with new characters or inserts them between the old characters. Unfortunately, there's no Insert LED on the keyboard.

It's important to realize that all the keys are the same, at least in Scan Code Set 3. Regardless of the keycap legend, each key produces a one-byte make code and a two-byte break code. How the PC interprets the codes is entirely up the program. If, for example, you'd like to have the Z key behave like the Caps Lock key and vice versa, it's a simple matter of software.

In most situations, however, it's a Good Idea to stay reasonably close to the PC standard. Once you see how it's done, though, you can twiddle the keyboard layout to suit yourself. Dvorak keyboard fans take note: it's just a few table entries away!

Ordinary real-mode BIOS functions return the scan code in AH and the character in AL. That leaves half of the 32-bit EAX register unused, which seemed a shame to me. I recycled another part of PC history by having KeyGtGetKey return the familiar BIOS shift-state bits in the high 16 bits of the register. Listing 3 shows the structure defining the bit layout.

Each locking shift key has two bits in that structure. One bit tells you the shift state: 0 for inactive and 1 for active. The other bit tells you if the key is currently pressed. In most cases,

you use only the first bit, but the second makes detecting oddball chords like Alt-NumLock-Insert-M a snap.

The Shift, Ctrl, and Alt keys have one bit apiece. Those six bits tell you when the corresponding key is pressed. In addition, Ctrl and Alt each have a summary bit that is active when either key is pressed. Most of the time, you use the summary bit, but it's easy to detect Left-Ctrl-Right-Shift-Q if you must.

The SysReq key is a bit of an oddball. The real-mode BIOS detects the make and break codes, then issues Int 15, AH=85 with AL=00 and 01, respectively. The FFTS keyboard interface simply flips the bit shown in Listing 3, although you could add whatever code you feel is appropriate for your setup.

The global variable ShiftState maintains the current state of the bits showninListing3. KeyDoSpecials detects keys that change the bits and updates them appropriately. Remember that the FFTS keyboard code calls KeyDoSpecials when each key is used, so the shift states track the most recent scan code removed from the ring buffer.

The keyboard LEDs should, of course, track the state of the locking shift keys. KeyUpdateLEDs in Listing 4 converts the three shift-state bits into the corresponding LED control bits, then sends the result to the keyboard. The command sequence is lengthy enough that I decided to cache the last LED command in a global variable and update the LEDs only when they change.

Now that you know how the shift keys work, we can explore the character keys.

SHIFTING STATES

KeyMakeChar is a truly hideous routine that boils down to a single, one-line table-look-up instruction. I'll explain it without listing it here because you can't tell what's going on just from looking at the code.

Listing 2 in the June column presents KeyCodes, the table holding all the information we need to convert Scan Code Set 3 characters into real-mode BIOS values. Basically, KeyMakeChar examines the shift bits to decide which column of that table is applicable, then uses the key's scan code as an index into the table. It sounds pretty straightforward.

Deciphering the shift states, however, is a Boolean nightmare.

The first column of KeyCodes produces what IBM calls ***base case,*** when all shift and lock keys are inactive. The remaining three columns hold the values when Shift, Ctrl, and Alt are active. As you might expect, the left and right keys in each pair produce the same result.

The BIOS prioritizes the various shift keys so that you can press any combination at once with any (or all] of the locking keys active and still get a character. Alt outranks Ctrl, which outranks Shift, all of which outrank the base case. If you press Ctrl-Alt-A, for example, you get the character code for Alt-A with the appropriate Ctrl and Alt bits set in the shift state.

FFTS takes a simpler tack, at least for the time being. For each character key, you may have only one shift key down in addition to any of the locking key states. This eliminates the multiply-shifted chords that come in handy at times. It also eliminates some fairly messy code that's best left until we actually need it and can do the debugging without too much extra effort.

For example, when Num Lock is active, the numeric keypad produces numbers if neither Shift key is active. Pressing Shift produces the cursor control key codes found on the Original PC keyboard, although it's easier to use the dedicated keys.

When Caps Lock is active, the letter keys produce capitals and all the other keys are unaffected. In this situation, Shift produces lowercase letters and upper-shift characters for everything else.

There are a few other twists and turns in the FFTS keyboard handler, but that should convince you that a perfect PC emulation is far from trivial. Download the code and see how your keyboard responds. The debugging information coming out the

serial port should give you plenty of insight into how things work.

HARDWARE REVISIONS

When I started on this topic, 1 found the keyboard interface on my '386SX had gone slightly sour. Given all the unsafe computing practices I do, it's a wonder the board lasted more than two years. On the other hand, not noticing a flaky keyboard for perhaps a year tells you something about the FFTS code we've been running!

1 replaced it with a 80486DX2/80 board from DreamTech, the folks who sold me the original '386SX. It dropped right in place with only a minor amount of twiddling. The new CPU clocks about 2400 task switches per second, roughly six times faster than the old board.

Yes, that does seem low for a 486-versus-386, dual caches versus no caches, nearly three-times clock ratio, double-width memory upgrade. I must write a column or two on performance one of these days.. As near as 1 can tell, don't believe the hype. As always, one careful measurement is worth 1,000 expert opinions.

FFTS worked perfectly after 1 tweaked a pair of delay loops to match the new CPU's speed. Shazam, all that oddball peripheral gear was up and running. Hooray for the PC Compatibility Barrier!

I also bought a new combination board with 16C550 UARTs. Can you diagram the two different pinouts for those ubiquitous 2 x 5 serial-port headers? If not, tuck Table 1 in your clip-and-save file. The new I/O board was, of course, different from the old one. I spent a pleasant afternoon tracking this down and making up a set of cables to match my screwball back-panel layout.

RELEASE NOTES

There's no new code this month, oddly enough, as last month's files had the complete, working keyboard interface.

In case you've wondered what it takes to build something like FFTS, here's the box score. All told, FFTS works out to 7,200 nonblank, noncomment assembler source lines. That's about 12,000 lines or 423,000 bytes of source code distributed in 41 files. In addition, each column has some disposable demo code, specialized boot sectors, PM loaders, and so forth and so on.

Not bad for a one-man, part-time effort, eh?

Next month, we lay the groundwork for Virtual-86 mode: running "real mode" 16-bit programs in 32-bit protected mode. ❑

***Ed Nisley** (KE4ZNU), **as Nisley Micro** Engineering, makes small computers **do amazing things. He's also a member of Circuit Cellar INK's engineering staff. You may reach him at** ed.nisley@circellar.com **or 74065.** 1363@compuserve.com.*

IRS

416 Very Useful
417 Moderately Useful
418 Not Useful