DEPARTMENTS

FIRMWARE FURNACE

Ed Nisley

Journey to the Protected Land: The Mystery of Scan Code Set 3

Just a little chat—with your keyboard that is. Building on the principles covered in last month's column, Ed shows us how to coax the PC into talking to the keyboard.

have you ever looked at one of your old projects and asked, "Who wrote this code and what was I thinking?" I know I have! Sometimes you can replace a page of tortured logic with a single, obvious, crystal-clear function.. .that is, until you look at it again in a few years.

Last month, you saw how the PC's keyboard evolved from a fairly simple subsystem into a complex mess. Each change made sense at the time, but the end result is essentially incomprehensible. Imagine designing a system that produces eight bytes for a single key.

This month, 1'11 examine the keyboard hardware and firmware built into every PC. Protected mode gives us the opportunity to switch the keyboard's fundamental operation into something sensible. As you'll see, talking to the keyboard exercises some interesting machinery.

Crystal clear? Check it again next year!

CONTROLLING THE CONTROLLER

The keyboard controller on the system board is an 8042 Universal Peripheral Interface, also known as a microcontroller. It's generally easy to spot since a 40-pin DIP looks terribly out of place on a system board where

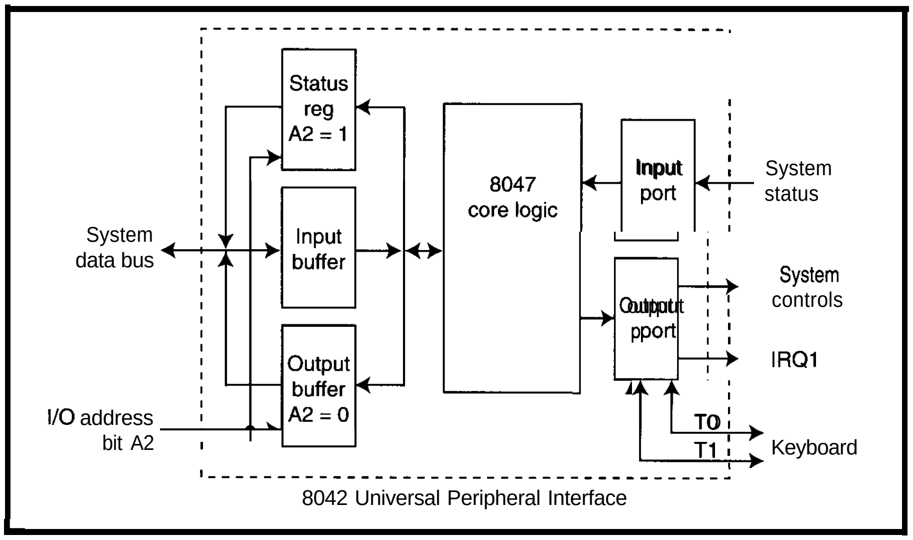

Figure 1—The system board keyboard controller is an 8042 microcontroller interfaced to the main CPU through a pair of byte-wide I/O ports at addresses 60 and 64. Port 64 sends commands to the controller and returns status bits; the values read and written are not identical. Port 60 is a bidirectional data port. The 8042 handles the keyboard's serial data format and serves as an interface to the system board's status and control bits.

three or four surface-mount LSI parts hold several million logic transistors. Current systems sport genetically engineered 8042 descendants stuffed with specialized speed-up hardware.

A great deal of weirdness surrounding the keyboard controller goes away as you read the Intel 8042 data sheet. Figure 1 sketches the key features. As you can see, all the logic is inside the 8042! The main system's only access is through two I/O ports at addresses 60 and 64 (hex) and the output driving IRQ 1.

Reading port 60 selects the 8042 Output Buffer through which the controller sends keyboard scan codes and other information to the '386SX. Reading port 64 selects the 8042 Status Register, which contains the five firmware-controlled bits and three hardware flags shown in Listing 1. Internal hardwired logic sets Status Register bit 0, the OBF flag, when the 8042 firmware writes a byte into the Output Buffer and clears it when the '386SX reads the byte.

The 8042 firmware controls an internal gate that routes OBF to the pin driving IRQ 1. Each time the firmware writes a byte into the Output Buffer, IRQ 1 goes high and the '386SX CPU executes the IRQ 1 interrupt handler. You can enable and disable this interrupt in three places: the 8042, the 8259, and the CPU's Interrupt Flag.

Writing a byte to either port 60 or port 64 loads the 8042 Input Buffer and sets Status Register bit 1, the IBF flag. Hardwired logic also copies I/O address bit 2 into Status Register bit 3 on each write, giving the firmware an easy way to distinguish the two sources. IBF goes low when the 8042 firmware reads the Input Buffer.

The keyboard controller firmware accepts data through port 60 and commands through port 64. Although the '386SX CPU writes to two separate ports, they neck down to a single chunk of 8042 hardware. You must verify that IBF is low before writing to either port, lest you overwrite a previous value before the 8042 has read it.

With that hardware background in mind, the keyboard controller should make more sense. The 8042 firmware recognizes about 120 commands written to port 64, most of which are entirely irrelevant for normal operation. Listing 2 shows the few commands used by the FFTS keyboard interface.

The terminology doesn't help much. The '386SX writes these commands to port 64. One of the commands, 60 hex, is *Write Command Byte.* The *Command Byte,* shown in Listing 3, is subsequently written to port 60. The only way to get familiar with this stuff is to use it.

Bit 6 of the Command Byte, Translate, determines whether the controller translates raw keyboard scan codes into system scan codes. This bit is normally on because the keyboard defaults to Scan Code Set 2, the baroque multibyte scheme I described last month. When Translate is zero, the controller passes the keyboard's scan codes directly to port 60 without modification. As I'll discuss later, turning this bit off is essential for the FFTS interface.

When bit 4, DisableKbd, is on, the 8042 firmware forces the keyboard clock line low to prevent the keyboard from sending anything to the controller. Normally, your set-up routine turns this bit on and your operating code uses the Enable and Disable Keyboard commands to flip the bit, thus eliminating the need to write a new Command Byte.

Setting bit 0, EnableInt, on enables the IRQ 1 interrupt whenever the OBF bit goes on. The hardware doesn't care why the Output Buffer became full; OBF goes on whenever the 8042 firmware writes a byte

Listing 1—You may read the keyboard controller status byte from port 64 at any time. The 8042 sets Output Full whenever a byte is available at port 60. This flag also triggers IRQ 1 if interrupts are enabled. The hardware sets Input Full immediately after each write to port 60 or 64, indicating that the firmware has not yet processed the byte. The controller sets ParityError, RecTimeout, and TransTimeout after performing several retries on its own.

```
RECORD                      STATFLAGS {
StatFlag_ParityError:1;  kbd serial parity error
StatFlag_RecTimeout:1     ; timeout during kbd message
StatFlag_TransTimeout:1   ; timeout after ctl message
StatFlag_NoKeylock:1      ; 0 = keylock switch ON
StatFlag_CmdReceived:1    ; 1 = last write to Port 64
StatFlag_SysFlag:1        ; keyboard OK or SysFlag = 1
StatFlag_InputFull:1      ; 1 = no write to 60/64
StatFlag_OutputFull:1     ; 1 = read data from 60
```

whether it's a scan code or a response to a command. You must ensure that your keyboard interrupt handler is never surprised by a byte that isn't a keyboard scan code.

The three routines shown in Listing 4 handle low-level system keyboard controller I/O. I have omitted statements that generate the tracing and debugging messages you'll see later.

Listing 2—*Most of the keyboard controller commands are not useful during normal operation. The FFTS routines use fhese few commands.* Listing 3 shows *the bits in the Command Byte.*

```
CCMD_RDCMD      = 020h     ; read controller command
CCMD_WRCMD      = 060h     ; write controller command
CCMD_TEST       = 0AAh     ; test controller
CCMD_DISABLE    = 0ADh     ; disable keyboard
CCMD_ENABLE     = 0AEh     ; enable keyboard
CCMD_SYSRESET  = 0FEh      ; reset the entire system!
```

KEYBOARD CONVERSATIONS

Notwithstanding the preceding discussion, the system keyboard controller's main purpose in life is converting between keyboard serial data and PC parallel data. Generally, our code talks directly through the keyboard controller to the keyboard itself, so we must know what the microcontroller under the keycaps expects to hear.

Unless the system keyboard controller is busy processing a command written to port 64, it simply passes data written to port 60 directly to the keyboard in serial form. The IBF flag indicates that the data previously written to either port 60 or 64 hasn't been processed yet. Listing 4b takes care of this with the same code that writes data to the Input Buffer after a system-keyboard-controller command.

Similarly, the controller converts any data arriving from the keyboard to parallel form and places it in the Output Buffer register, triggering IRQ 1 if interrupts are enabled. Reading port 60 extracts the data, resets OBF, and clears IRQ 1. The controller disables the keyboard while OBF is set, eliminating the possibility of an overrun.

Listing 4c reads Output Buffer bytes without regard to where they came from and thus returns bytes from the keyboard as well as the system keyboard controller. This code is useful only after commands producing a response byte because the normal keyboard scan codes should go to the IRQ 1 interrupt handler.

Serial data flows between the system keyboard controller and the keyboard at a peak rate of about 10 kbps (see Photo 1, FF, *INK 59).* The average data rate is much lower, limited mainly by delays between the bytes-even the fastest typists have trouble generating a few hundred keystrokes per second. I'd say the link is fast enough.

The communications protocol's details aren't of much use to us right now. Suffice it to say that when the system keyboard controller lowers the clock line, the keyboard cannot transmit information and stores keystrokes in its internal buffer. You must disable the keyboard using the system keyboard controller Disable Keyboard command (AD hex) before issuing any commands using the Input or Output Buffers. If the keyboard remains enabled, it may send a byte that arrives just after your command and confuse the proceedings.

The keyboard sends an acknowledgment for every byte it receives from the system. If the byte had good parity and timing the keyboard sends FA hex (pronounced "Ack"). Most errors results in FE hex [say "Error"). Mercifully, the system keyboard controller handles error conditions by resending the byte several times before setting the Status Register error bits.

Listing 5 shows how to send a byte to the keyboard and process the acknowledgment. I suspect the only proper response to an error that makes it past the system keyboard controller's retries is "Your keyboard just died." In this routine, I simply ignore persistent errors and continue without complaint.

The keyboard recognizes about a dozen commands, many of which aren't relevant to our purposes. Listing 6 shows the few we'll need for the FFTS routines.

Now onto the fun part!

CONVERTING THE CODES

The complexity of deciphering all the scan codes produced in all the shift states for all the keys seemed too daunting when I first looked into this topic. Frankly, writing a replacement BIOS keyboard handler wasn't something I wanted to tackle!

When faced with an impossible situation, sometimes you can restate the problem so the solution is obvious. In this case, a light went on when I read that PS/2 and some other key-

Listing *3-This Command Byte defines fhe controller's overall operating mode. If the* system *includes a F'S/P-style system-board mouse port, several of the Command Byte bits have different meanings. The B/OS sets Tr a n s 1 a t e, Sys F l a g, and En a b 1 e In t affer each system reset. The FFTS keyboard routine turns Jr an s l a t e off to* gain direct access *to* the keyboard's *scan codes.*

```
           RECORD CTLFLAGS {
           CtlFlag_Res7:1=0              ; reserved, must be 0
           CtlFlag_Translate:1=1         ; translate AT to PC
           CtlFlag_PCMode:1=0            ; use PC serial intf
           CtlFlag_DisableKbd:1=0        ; force kbd clock low
           CtlFlag_DisableInhibit:1=0    ; override keylock
           CtlFlag_SysFlag:1=1           ; 1 after BIOS setup
           CtlFlag_Res1:1=0              ; reserved, must be 0
           CtlFlag_EnableInt:1=0         ; enable IRQ 1 interrupt

CMD_NORMAL = (MASK CtlFlag_SysFlag) + MASK CtlFlag_EnableInt
```

Listing 4-a) The KeySendCmd routine waits until the 8042 Input Buffer Full flag is clear, then writes the command to port 64. b) KeySendData waits until both the Input and Output Buffer flags are clear, then writes the data byte to port 60. Any scan codes arriving before KeyFlushOutput are lost. c) The KeyReadData routine polls the Status Register until the Output Buffer flag goes high, then reads the byte. The keyboard interrupt handler snags the byte first unless you disable IRQ1.

a)
```
    PROC   KeySendCmd
    ARG    CmdByte:DWORD
    USES   EAX

    CALL   KeyWaitInBuff          ; wait for prev cmd to finish

    MOV    EAX,[CmdByte]          ; fetch the command
    OUT    KEY_CMD,AL             ; send it out

    RET
    ENDP   KeySendCmd
```
b)
```
    PROC    KeySendData
    ARG     DataByte:DWORD
    USES    EAX,ECX

    CALL    KeyWaitInBuff         ; wait for cmd to clear
    CALL    KeyFlushOutput        ; discard any pending bytes

    MOV     EAX,[DataByte]        ; fetch the data
    OUT     KEY_DATA,AL           ; send it out

    RET
    ENDP    KeySendData
```
c)
```
    PROC       KeyReadData
    USES       ECX

    MOV        ECX,MAX_DATAWAIT   ; maximum delay

@Stall:
    IN         AL,KEY_STATUS      ; key ready yet?
    TEST       AL,MASK StatFlag_OutputFull
    JNZ        @Fetch

    CallSys    CGT_TMR_DELAYMS,1; nope, wait a bit
    LOOP       @@Stall            ; and try again

@Fetch:
    IN         AL,KEY_DATA        ; fetch the data
    Punt
    MOVZX      EAX,AL             ;clean up the byte
    MOV        [LastCode],EAX     ; and save for examination

    RET
    ENDP       KeyReadData
```

boards supported three scan-code sets, one of which produced a single byte per keystroke. I knew a bit about why Scan Code Set 3 existed, a tale told here last month. The only remaining question was how many clone keyboards supported that feature.

Rick Freeman and the folks at Computer Options here in Raleigh graciously loaned me one of every keyboard in the store. Adding those to my ragtag collection, I checked out a dozen Enhanced keyboards and found that, with a single exception, they all supported Scan Code Set 3. The oddball, a Northgate C/T keyboard, dates back to 1988 when Enhanced keyboards were very, very new.

While that's not conclusive proof that all PC keyboards respond cor-

FREE
Data Acquisition Catalog

1995

PC and VME data acquisition catalog from the inventors of plug-in data acquisition.
Featuring new low-cost A/D boards optimized for Windows, DSP Data Acquisition, and the latest Windows software.
Plus, informative technical tips and application notes.
Call for your free copy
1-800-648-6589

ADAC
American Data Acquisition Corporation
70 Tower Office Park, Woburn, MA 01801
phone 617-935-3200 fax 617-938-6553

#117

rectly to the code in FFTS, it gave me enough confidence to try this trick. I'm interested in hearing how this works out on your system. But, if your keyboard doesn't support Scan Code Set 3, you get to decode its output. I want no part of Scan Code Set 2, thank you very much.

The code in Listing 7 handles the transition from the BIOS default keyboard settings to the new conditions. It runs the self-test routines in the system keyboard controller, and the keyboard then sends several commands to the keyboard. If any of the first few commands fail because of a missing keyboard or missing feature, the keyboard is disabled and unusable in FFTS.

I always set my keyboards for the shortest typematic delay (250 ms) and the fastest repeat rate (30 characters per second). Recognizing that your reflexes may vary, I've declared a group of constants that go all the way to stun: two characters per second after a one second delay.

Scan Code Sets 1 and 2 automatically set all keys to typematic-make-break mode, leaving the BIOS to sort out the superfluous make codes. Scan Code Set 3 works differently, placing only the typewriter and cursor keys in typematic mode. Most of the remainder are make-only keys that send a single code when they're pressed, do not repeat no matter how long they're down, and do not send a break code when they're released.

Most of the shift keys operate in make-break fashion, sending only a single make code and a single break code. Oddly enough, the right-Alt and right-Ctrl keys are make-only, which is sensible when you see the mainframe and minicomputer keyboard keycaps: one is an Enter key and the other does something similar. Remember, PCs aren't the only computers in the world!

The good news is that you can reprogram the key modes to suit your needs. A single command sets all the keys to the familiar typematic-make-break mode used in the other Scan Code Sets. This mode is appropriate for typewriter keys, cursor keys, and a few others, eliminating the need to reprogram each and every key individually.

Three additional commands set individual keys to make-only, make-break-only, or typematic-only mode. The key's make scan code follows the command byte, which means you must program each key individually. The keyboard accepts these commands when any Scan Code Set is active, but they apply only to Scan Code Set 3.

The make-break mode is a natural for shift and lock keys as the make code indicates that the key is down and the break code says it's up. There is no need to process and discard additional make codes.

The most useful function keys in make-only mode are Esc, Ins, Home, and End. These keys produce a single scan code that triggers a single action. Again, not having to deal with repeated keys simplifies programming.

Some applications can probably take advantage of typematic-only function keys that send repeated make codes with no break code at the end. All the FFTS definitions reside in a table, making it easy to contort the keyboard to suit your needs.

Listing 5—*The keyboard acknowledges each byte if receives with either FA hex (good) or FE hex (error). This routine resends the byte a few times and then simply ignores the error. You must disable the keyboard before calling this routine to ensure that KeyFlushOutput doesn't inadvertently discard a keystroke.*

```
        PROC    KeySendDataAck
        ARG     DataByte:DWORD
        USES    ECX

        MOV     ECX,MAX_RETRIES       retry counter

@Resend:
        CALL    KeyWaitInBuff         wait for cmd to clear
        CALL    KeyFlushOutput        discard any pending bytes

        MOV     EAX,[DataByte]      ; fetch the data
        OUT     KEY_DATA,AL         ; send it out

        CALL    KeyReadData         ; fetch byte
        MOVZX   EAX,AL              ; clean it up
        CMP     AL,KRSP_ACK         ; ack?
        JE      @Done               ; yes, done
        CMP     AL,KRSP_RESEND      ; resend?
        JNE     @@Done              ;  no. ignore it
        LOOP    @Resend             ; yes, try again

@@Done:
        RET
        ENDP    KeySendDataAck
```

Listing 6—*The keyboard responds to a variety of* **commands** *sent through the system keyboard controller. Some systems include an onboard mouse port driven by the system keyboard controller; the mouse controller responds to a slightly different command set. This list includes the most useful keyboard commands.*

```
KCMD_WRLEDS     = 0EDh  ; write to keyboard LEDs
KCMD_CODEMODE   = 0F0h  ; get/set scan code mode
KCMD_RDID       = 0F2h;  read keyboard ID
KCMD_RATE       = 0F3h  ; set typematic delay & rate
KCMD_ENABLE     = 0F4h  ; enable keyboard
KCMD_ALL_TMB    = 0FAh  ; all keys typematic/make/break
KCMD_ONE_T      = 0FBh  ; set single key to typematic
KCMD_ONE_MB     = 0FCh  ; set single key to make-break
KCMD_ONE_M      = 0FDh  ; set single key to make-only
KCMD_RESET      = 0FFh  ; reset to power-on defaults
```

Listing *I-The* protected-mode ***FFTS*** keyboard interface uses Scan *Code Set 3 because if's much easier* ***to*** *process than* ***the PC*** *default, Scan* Code ***Set*** 2. *This routine* ***tests the*** *system keyboard controller and* ***the*** *keyboard, sets* ***the*** *keyboard's new operating modes, and prepares the interrupt handler. If the keyboard isn't present or doesn't support Scan Code Set 3, the code* ***displays*** *a message and* ***doesn't*** *enable* ***the*** *keyboard.*

```
 MOV    [KeyEnable],1                             assume  OK..

 CALL  KeySendCmd,CCMD_TEST                       test  controller
 CALL  KeyReadData                                fetch  result  code
 CMP    AL,055h                                   is  it  OK?
 JE     @@CtlOK
@@KbdNG:
 MOV    [KeyEnable],0                             keyboard  failure!
 CALL  ConfSendString,CON_SERIAL,  \
        GDT_CONST,OFFSET cMsg_KbdNG
 JMP    @@Done

@@CtlOK:
 CALL   KeySendCmd,CCMD_DISABLE                   disable  keyboard
 CALL   KeyFlushOutput                            discard  pending  codes

 CALL  KeySendDataAck.KCMD_RESET                  reset & test keyboard
 CMP    AL,KRSP_ACK                                accepted byte?
 JNE    @@KbdNG                                    nope,  no keyboard
 CALL  KeyReadData                                fetch  result  code
 CMP    AL,OAAh                                   is  it  OK?
 JNE    @@KbdNG                                    nope,  bad  hardware

 CALL  KeySendDataAck,KCMD_RATE                   set  typematic  rate
 CMP    AL,KRSP_ACK                                did it work?
 JNE    @@KbdNG                                 ;  nope,  bad  hardware
 CALL KeySendDataAck,DELAY_250 + RATE-30  ; yup,  get  smokin'

 CALL  KeySendDataAck,OFOh                        select  scan  code...
 CALL  KeySendDataAck,003h                         Set  3
 CMP    AL,KRSP_ACK                               did it work?
 JNE    @@KbdNG

 CALL  KeySendDataAck,KCMD_ALL_TMB                all  typematiclmlb
 CMP    AL,KRSP_ACK                               did it work?
 JNE    @@KbdNG

 CALL   KeySetTypes                               set  individual  keys

 CALL  KeySendDataAck,KCMD_ENABLE                 enable  key  scanning

;--- install the IRQ 1 interrupt handler

 CallSys  CGT_MEM_SETINTGATE,I8259A_VECTOR_PM+1, \
          GDT_IDT_ALIAS, \
          GDT_CODE,<OFFSET KeyHandler>,ACC_INTGATE

;--- enable 8259 keyboard controller interrupt on IRQ1

 CallSys  CGT_UTIL_UNMASKIRQ,1

;--- set numlock on, update the LEDs, and (en passant)
;--- enable the keyboard

 OR     [ShiftState],MASK KEY_SH_NUMLOCK
 CALL   KeyUpdateLEDs
```

Each of the routines called in Listing 7 sends tracing information to the serial port. If the keyboard can't support Scan Code Set 3 or if it reports a self-test error, you'll see a few additional messages. The self-test and keyboard response delays differ among the keyboards I've tested. I've picked default timeouts long enough to handle much worse than the worst I've seen.

CPL	Current Privilege Level
DPL	Descriptor Privilege Level
EOI	End Of Interrupt (command)
FDB	Firmware Development Board
FFTS	Firmware Furnace Task Switcher
GDT	Global Descriptor Table
GDTR	GDT Register
IBF	Input Buffer Full
IDT	Interrupt Descriptor Table
IF	Interrupt Flag
IOPL	I/O Privilege Level
LDT	Local Descriptor Table
LDTR	LDT Register
NT	Nested Task
OBF	Output Buffer Full
P bit	Present bit (in a PM descriptor)
RF	Resume Flag
RPL	Requestor Privilege Level
TF	Trap Flag
TR	Task Register
TSS	Task State Segment

The end result of all this is a sensible keyboard-each key has a unique, single-byte scan code. We can surely build something interesting from that raw material!

RELEASE NOTES

Demo Taskette 3 now displays doublewords containing the shift state, system scan code, and character for each keystroke that produces a character. The keyboard interface routines send a torrent of tracing information to the serial port on each make and break code, exposing the inner workings (and perhaps failings) of your keyboard.

I planned to wrap up the keyboard this month, but there's more code than pages. Next month, we'll look at the interrupt handler that queues scan codes and the translation routines that convert them into familiar real-mode BIOS values. ❑

Ed Nisley, 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

422 Very Useful
423 Moderately Useful
424 Not Useful