Patching hardware interrupts • maximise system control

Hardware interrupts differ from software interrupts in two important respects—they can happen unexpectedly, and the hardware concerned usually needs some form of finishing process to clear the interrupt. The simplest one we can deal with is the keyboard hardware interrupt, INT 9, which we'll use in this example to catch the CapsLock key.

The example program called BELAYCAP requires you to hit the CapsLock key twice in rapid succession before it allows the keystroke through to the BIOS and so set the CapsLock on. This example program will be useful if you have the habit of inadvertently hitting the CapsLock key then typing in capitals. It can also act as a template for future programs that intercept the keyboard.

As in last month's program, the first thing the program does is to jump to the initialisation code at the end. Here, we patch our code in to the keyboard hardware interrupt and the software timer interrupt. Another software timer interrupt is INT 8, which the BIOS uses to generate the one we are patching: INT 1CH. The reason we don't patch INT 8 is that some programs increase the PC timer frequency from the standard 18.2 times per second so that they can check their tasks more frequently. This patched, faster clock is then divided down in software, and is still passed on to INT 1CH at 18.2 times per second.

So, our timer routine is called on a regular basis. The timer routine checks to see if we have counted to zero already and if we have, passes control to whatever the old timer routine was. If the count isn't zero, it is decremented and checked again. If the result is zero this time, the variable 'state_rtn' is set to its default value, before control is passed to the original timer routine. The purpose of 'state_rtn' will become apparent later, but the net effect is that 'state_rtn' is set to the default value when the count expires. This is one way of implementing a timer.

Back at the keyboard interrupt, the first thing we do is save the AX and DS registers. These are needed to look into the BIOS variable area.

From our BIOS variables list, we know that the state of the CapsLock key is held in bit 6 of byte 40:0017H. If this bit is already set, then we know that CapsLock is already active and that the user is trying to turn it

BELAYCAP: Program activated by keys

```
; BELAYCAP: Program which requires the CapsLock key to be hit
; twice to activate.

INCLUDE     INTNOS.DEF

caps_key equ 58      ; Scan code for caps key

code SEGMENT public 'CODE'
     ASSUME cs:code,ds:code        ; No data segment

     org 100H

begin:
                     ; Jump to init code
     jmp     init_bit


;=============================================
; This is our diverted interrupt. It corrupts nothing.
;=============================================
diverted_kbd_int    PROC    far

     ASSUME ds:nothing

     push    ds
     push    ax
     mov     ax,40H ; Check key flags in BIOS
     mov     ds,ax
                     ; Test if caps active.
     test    byte ptr ds:[17H],40H
     jnz     dki_abort       ; Jump if active
;
; See what manner of key has been pressed.
;
     in      al,60H ; Kbd data port
     mov     ah,al  ; Keep original in AH
     and     al,7FH ; Mask off top bit
                     ; Was it our caps key?
     cmp     al,caps_key
                     ; Jump if not.
     jnz     dki_abort
;
; It was a caps key. What should we do with it?
;
     call    cs:[state_rtn]
                     ; Jump if we pass it on
     jc      dki_abort
                     ; Otherwise get rid of it
;
; We need to absorb this key. Reset the keyboard
; port, issue a non-specific End of Interrupt,
; and return.
;
     in      al,61H ; This resets the keyboard
     mov     ah,al  ; port so that the caps
     or      al,80H ; lock keypress is lost.
     out     61H,al
```

(continued)

off. In this case, we must let the key codes through whatever they are.

If the CapsLock isn't already on, then we read data from the PC keyboard port using the IN instruction. This command looks at specific addresses to which hardware is connected and comes in several forms: IN AL,nn, IN AX,nn, IN* AL,DX, IN AX,DX, INSB, and INSW. As hardware can sometimes be a little slow, IN instructions can take much longer than an equivalent-sized read of a memory location.

The instruction 'IN AL,nn' reads the byte value at hardware location (or input port) nn into register AL. Note that this can only be used for port addresses where nn is less than 100H. IN AX,nn is similar, but puts the value of the hardware port at address nn in AL, and then reads nn+1 into AH. This instruction is used on processors above the 8086 to do such things as read data rapidly from a hard disk controller.

IN AL,DX and IN AX,DX do a similar thing, but allow addresses in the range 0-FFFFH to be used in DX. Most PC peripherals are at addresses of lower than 3FFH; the keyboard port that we are interested in is at 60H.

INSB and INSW are input port instructions allied to the string commands STOSB and STOSW. They are used in exactly the same way, but take their source from the port address in DX. They don't exist on the 8088 and 8086 processors.

With the key scan code now read from the keyboard into AL, we can set about seeing what to do with it. Key scan codes are not ASCII, but are arbitrary numbers assigned to the keys. A full list can be found in good manuals, or you could write a program to display them as an exercise.

Before checking the value, remove the top bit. This is because the key scan code is a 7-bit quantity, and the top bit is clear when the key has just been pressed, and set when the key has been released. If the scan code isn't a CapsLock, we pass it on to the BIOS. If it was a CapsLock key, we pass it on to whatever routine 'state_rtn' happens to be pointing at.

It's possible to write some unnecessarily complex code that does a different task every time it's entered. Usually, it tests which bit of it should be running, and after many tests gets to the right piece. An easier way is to make each routine store a pointer (or an index) to the next routine that should be executed. If the current routine has to branch, then it just stores the pointer to the bit it wants to branch to, and this will be called next time round. This process is called a state machine. Programmers and hardware designers alike often find it helpful to draw out maps of the interconnected states, much like flowcharts.

(continued)

```
        xchg    ah,al
        out     61H,al

        mov     al,20H ; Non-specific end of
        out     20H,al ; interrupt command

        xor     al,al           ; Clear carry
        jmp     short dki_done

dki_abort:
        stc
dki_done:
        pop     ax
        pop     ds
;
; If carry is set here, we do not want to prevent
; the key from going to the BIOS routine.
;
        jc      dki_pass_on
;
; We have processed this key already. Return from
; interrupt
;
        iret

dki_pass_on:
        jmp     cs:[orig_kbd_vec]

; End of the diverted interrupt.
diverted_kbd_int    ENDP

;=============================================
diverted_timer_int PROC    far
; Decrements our own tick count when the system
; calls it every 54ms (18.5 ticks is just about
; a second). When we hit zero, we reset the state machine.
;=============================================

                        ; Is our tick count zero?
        cmp     cs:[tick_count],0
                        ; Jump if it is already.
        jz      dti_skip_dec
                        ; If not, dec it again.
        dec     cs:[tick_count]
                        ; Jump if still not zero
        jnz     dti_skip_dec
;
; We hit zero, Abandon any attempts to set caps.
;
        mov     cs:[state_rtn],offset caps_key_wait
;
; Whatever happened, now do the original timer int.
;
dti_skip_dec:
        jmp     cs:[orig_timer_vec]

diverted_timer_int ENDP

;=============================================
caps_key_wait       PROC    NEAR
; This routine waits for our first caps lock.
; It starts off our timer, and sets up the next routine.
;=============================================
```

(continued)

Our state machine is fairly simple: in the initial state, it waits for a CapsLock down scan code. If it gets an up code, this is passed on. When it finds a down code it sets the timer discussed above and invokes state 2. State 2 waits until it sees a CapsLock key coming up. When it does, it switches the machine to state 3. The key code is absorbed whether it is coming up or going down.

State 3 passes on the next CapsLock key code and resets the machine back to state 1. Don't forget that while all this is going on, the timer is ticking away. When it expires, everything is set back to state 1 again.

After the state machine has been called, the state of the carry flag is checked to see if the key scan code should be absorbed or not. Without going into too much detail of the PC's internals, the collection of IN and OUT instructions to 61H tell the keyboard controller to forget it had a scan code.

OUT instructions are the reverse of the IN instruction, and there are obvious similarities. The OUT family consists of: OUT nn,AL, OUT DX,AL, OUT DX,AX, OUTSB and OUTSW. The same rules apply to the values of nn and DX.

The final OUT 20H,AL tells the PC's interrupt controller to clear the interrupt, and then we drop into popping AX and DS off the stack. Finally, depending on the state of the carry flag, we either pass on the key to the original keyboard interrupt vector, or issue an IRET ourselves.

As the main purpose of writing short interrupt handlers is to patch an existing interrupt, readers may find the XLAT command useful. This takes the value in AL together with the value in BX, and replaces the value in AL with the byte at [BX+AL]. This is obviously just the job for 'look-up tables' as you would use for replacing, say, foreign characters when patching a printer interrupt.

The value in AL is taken as an unsigned value, so when AL is 0FFH the byte at [BX+0FFH] will be retrieved rather than the byte at [BX-1].

For indexing into other segments, a dummy parameter is used for no other purpose than to give the assembler something to relate a segment to. The instruction then takes the form XLAT ES:[BX], where ES can be any segment register. Some assemblers use the XLATB opcode when there's no following dummy parameter.

Common Code

Throughout this series we've accumulated a fair number of useful routines that are common between different programs. It's common assembler practice to separate these out into include files. If you make sure that you use the same constants

(continued)

```
     and    ah,80H ; If top bit is set, key is coming up.

     stc           ; Assume we pass on.
jnz  ckw_exit; So pass it on.

;
; Wait for caps key for just over 1/2 sec.
;
     mov    cs:[state_rtn],offset caps_rel_wait
     mov    cs:[tick_count],10
     clc           ; Don't pass on key.
ckw_exit:
     ret

caps_key_wait       ENDP

;=============================================
caps_rel_wait       PROC   NEAR
; Waits for the caps key to be released. We always
; absorb the key.
;=============================================

     and    ah,80H ; See if it is coming up.
     jz     crw_exit; Jump if not coming up.

                   ; Prepare to pass on next key
     mov    cs:[state_rtn],offset caps_pass
crw_exit:
     clc           ; Don't pass on this key.
     ret

caps_rel_wait       ENDP

;=============================================
caps_pass   PROC   NEAR
; Passes on the next caps lock key and puts our
; state machine back to the original state.
;=============================================

     mov    cs:[state_rtn],offset caps_key_wait
     stc    ; Pass on the key
     ret

caps_pass   ENDP


; This is the original interrupt vector.
orig_kbd_vecdd      ?
; This is the original timer vector
orig_timer_vec      dd     ?

; Used for our own timing routines
tick_count  dw      0
; Keeps track of current caps subroutine
state_rtn   dw      offset caps_key_wait

init_bit:
;=============================================
;           "Throwaway Code"
; This is the bit that initialises the interrupt
; patch. It must be at the end, as space is
; reserved from the start of the program.
;=============================================

    push    cs      ; Get our data segment
```

(continued)

throughout all the code, there's less opportunity for confusion. By convention, include files which define constants are given the extension DEF, and ones defining macros (more on these later) are given the extension MAC. Our include file INTNOS.DEF looks like this:

```
; INTNOS.DEF —Common interrupt
; numbers
keybd_hw_int equ     9
video_int    equ     10H
disk_int     equ     13H
timer_int    equ     1CH
dos_int              equ     21H
```

We can add more interrupts as we go along. To include these in our assembler code, we use the line:

```
INCLUDE INTNOS.DEF
```

C programmers should note that the include file name is generally not enclosed in quotation marks.

Where the include files are stored is a different matter. They can be kept in the same directory as the other assembler files (.ASM, .A86, or similar) but as they are likely to be shared across several programs, there's usually a way of defining a directory for include files. This varies for the different assemblers, but under Microsoft's MASM and Borland's TASM you would use '/IC:\INCLUDES' to use the INCLUDE directory on drive C:.

Final thoughts

After this sixth tutorial, writing assembler code should get easier. But there are bound to be routines that are easier in assembler and some that are best written in a high-level language such as C. For instance, a serial port interrupt handler is probably best written in assembler, but the rest of the code may not be machine specific. It's possible to write assembler code that interfaces with high-level languages in one of several ways, but there are a few things that you must establish either from the manual that comes with the language, or from the appropriate technical support department.

First, find out which registers are supposed to be preserved. Some need to be, and forgetting them is a common way of fouling up a machine code interface. It would also be nice to know the register conditions on entry to your own code, although if pressed you assume nothing.

Second, find out how you're expected to return from your assembled code to the main program. Some will demand a far return, some a return from interrupt and some a near return. Most likely, it will be a far return, but check on it.

Finally, you need to know the format of any data being passed to you, and the for-

(continued)

```
        pop     ds
        ASSUME  ds:code

                        ; Announce ourselves
        mov     dx,offset signmsg
        mov     ah,9
        int     21H

        push    es      ; Save our PSP

; Read the original keyboard interrupt

        mov     ax,3500H+keybd_hw_int
        int     21H

; Store the address returned in ES:BX

        mov     word ptr ds:[orig_kbd_vec],bx
        mov     word ptr ds:[orig_kbd_vec+2],es

                        ; Patch the interrupt

        mov     ax,2500H+keybd_hw_int
                        ; Put our routine address
                        ; in DS:DX
        push    cs
        pop     ds
        mov     dx,offset cs:diverted_kbd_int
        int     21H

; Read the original timer interrupt

        mov     ax,3500H+timer_int
        int     21H

; Store the address returned in ES:BX

        mov     word ptr ds:[orig_timer_vec],bx
        mov     word ptr ds:[orig_timer_vec+2],es

                        ; Patch the interrupt

        mov     ax,2500H+timer_int
                        ; Put our routine address
                        ; in DS:DX
        push    cs
        pop     ds
        mov     dx,offset cs:diverted_timer_int
        int     21H

        pop     es      ; Recover PSP

        mov     dx,offset cs:init_bit
        mov     cl,4
        shr     dx,cl

; Add one para of memory for rounding errors
; For .EXE versions, add 11H to account for the PSP
        inc     dx
                        ; Terminate but stay resident
        mov     ax,3100H
        int     21H

signmsg     db      13,10
        db      "BELAYCAPS V1.0 —Requires Caps Lock to "
        db      "be hit twice rapidly to activate."
        db      13,10,"$"
code ENDS

        END     begin
```

Available as:BELCAP.ZIP in Productivity Library (#3), in PCMAGUK forum. To access GO ZNT:PCMAGUK

mat in which you are expected to pass it back. This is fairly straightforward using C and integer or long arithmetic, but with BASIC, the format is much more complicated and implementation specific.

Bear in mind that many compiled languages have 'model' options to determine the number of segments that may be used to keep code and data in. With 'C', small code models usually require an ordinary RTE, but large ones a REEF to return. Likewise, small data models assume a fixed data segment, but large models assume data is 'far' and supply a segment:offset pointer to it.

Some languages—notably Borland's C—allow in-line code to be generated. This is either done by specifying the values of the bytes, or by writing assembler mixed up with the high-level C language.

A more sophisticated and flexible way is to assemble the language and assembler code to OBJ files and to combine these with a linker. Most compiled languages come with example programs that are a convenient template for producing your own code with, often including such features as automatically adjusting the return instructions and parameter pointers as the code changes from small to large models.

Parameters—far pointers, words or whatever—tend to be passed on the stack. Byte-sized parameters are usually converted into an even number of bytes so that the stack is always on an even boundary. It's important to do this, or the processor will have to access one word of memory for the high bytes of the stack and one for the low bytes. This slows down performance noticeably.

Assuming that we have a series of parameters passed on the stack, we use the BP register (which automatically indexes to the stack segment) to access them. First, as when changing interrupt flags, BP itself is pushed onto the stack. Assuming a far return, our 'stack frame' will look something like this:

```
     BP      ->      Original
value of BP
     BP+2    ->      Segment to
return to
     BP+4    ->      Address
within return segment
     BP+6    ->      Parameter
1
     BP+8    ->      Parameter
2
     BP+10   ->      Parameter
3
     ...             ...
```

On exit, pop BP off the stack (after any others we may have preserved), and return. If the routine requires a near return, then the BP+2 line will not exist, and everything moves back 2 bytes—a good reason to use the templates provided with the language. If you disassemble compiled languages, you may find code like:

```
PUSH    BP
MOV     BP,SP
SUB     SP,10
...
...
...
MOV     SP,BP
POP     BP
RET
```

This is preserving space on the stack for use within the routine. The space can be accessed as [BP-2],[BP-4] and so forth. This is a convenient, if slow, method of reserving memory provided you have sufficient stack. Each invocation of the routine reserves space for its own local variables, so eliminating re-entry problems.

Next month, we'll take a look at structures and macros in assembler, and how they help with stack frames. If you'd like to try to write a program as an exercise, try writing a screen saver that sets all the screen attributes to black on black if no key has been pressed for a while—restoring them when the next key is pressed. ■