9

Interrupt Processing

The hardware interrupt is a powerful mechanism that provides support for many of the services the operating system supplies. As described in Chapter 2, a device requests interrupt service by sending the CPU a signal on its interrupt line. Before executing an instruction, the CPU checks the interrupt line, and ‘‘calls’’ a procedure to handle the interrupt if it finds one pending. When the called routine ‘‘returns,’’ control passes back to the process that was executing, as if nothing had happened. Without an interrupt mechanism, the operating system could not guarantee that it would ever regain control once it started executing a user process.

Before looking at devices in detail to see why they interrupt and how the system responds, we will explore in this chapter the routines PC-Xinu uses to field interrupts and pass control to the appropriate routine. Later chapters explain more about the clock and keyboard devices, showing how interrupt processing routines are designed.

9.1 Dispatching Interrupts

We said that the hardware calls the interrupt handler when it finds an interrupt pending. The terms *call* and *return* have special meaning when applied to interrupts. First, there is no procedure call instruction in the interrupted program – the processor simulates a call to the appropriate interrupt routine ‘‘in between’’ the execution of normal instructions in the user’s program. Second, the processor automatically saves the current FLAGS and CS:IP by pushing them on the stack when an interrupt occurs (the interrupt routine must save and restore any other registers that it needs to use). Third, the interrupt routine must use a special return instruction to pop the old CS:IP and FLAGS off the stack in a single step.

Because interrupt routines manipulate hardware registers and use special call/return sequences, they cannot usually be written in high-level languages like C. The temptation is to write all the code related to interrupt processing, a significant portion of the operat-

ing system, in assembly language. But writing in a low-level language makes the software difficult to understand or modify. So, to keep the system code as readable as possible, our design employs a two-level strategy for processing interrupts. Interrupts branch to a small, low-level *interrupt dispatch* routine that is written in assembly language. The dispatcher handles tasks like saving and restoring registers, identifying the interrupting device, and returning from the interrupt when it has been processed. However, it does little else – it calls high-level routines to do the real work of interrupt processing, passing them enough information to identify the interrupting device.

PC-Xinu handles interrupts from the clock, the keyboard, and pseudo-devices such as *Ctrl-Break* described in Chapter 2. It also handles exception interrupts. This chapter concentrates on the dispatcher, deferring a discussion of most of the actual PC-Xinu interrupt service routines until later chapters.

9.2 The Interrupt Dispatcher

Devices connect to the computer system through hardware mechanisms called controllers. Controllers, which may be as simple as a chip or as complicated as a microprocessor system, usually reside on boards plugged into the system bus. They have hardware that translates between digital data and the signals necessary to control and communicate with peripheral devices like keyboards and disks, as described in Chapter 2.

Some devices, such as terminals, allow simultaneous transfer of data in both directions and consist of two independently controlled devices, one for input and one for output. These devices often employ one controller for both input and output. The controller may use separate vectors for input and output interrupts or (as is the the case for the PC communications devices) one vector for both input and output interrupts. In the latter case, it may be necessary for the software to distinguish between an input interrupt and output interrupt by interrogating the controller itself, but at the lowest level, the system must operate as if the input and output devices were independent.

9.2.1 The Intmap table

PC-Xinu uses an interrupt table, called *intmap*, which has one entry for each device requiring interrupt service by the operating system. An *intmap* entry contains information about the device together with a CALL instruction to the common interrupt dispatcher *intcom*. The interrupt vector for a device points to the CALL instruction in its *intmap* entry, so that when an interrupt occurs, the processor executes the CALL instruction which immediately executes the code in the dispatcher. Figure 9.1 shows the logical organization.

If all interrupts call the same dispatch routine, how does the dispatcher know which high-level interrupt routine to call? Remember that the CALL instruction pushes its return address on the stack just before branching to the called code. When the processor executes the CALL instruction in the *intmap* table, the address of the byte following the

instruction is left on the stack. The *intcom* code can use this address as a pointer to the *intmap* entry for the device, allowing access to information, including the address of the device's high-level interrupt routine and other device-specific parameters.

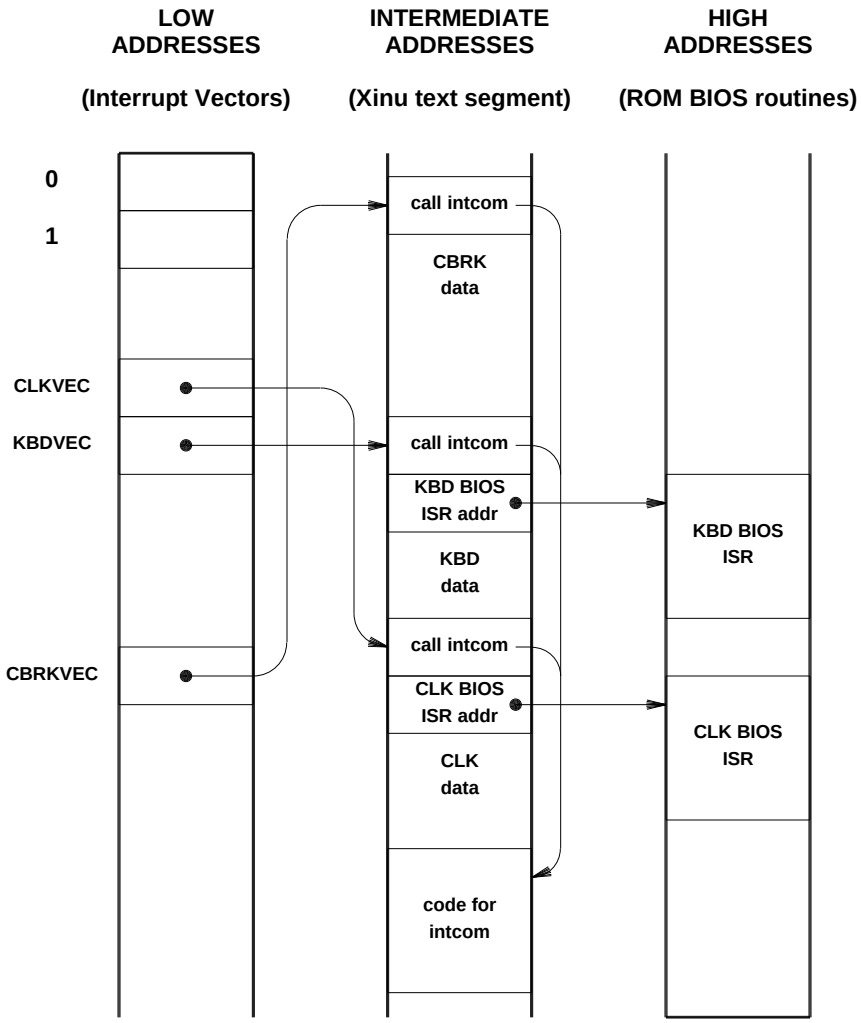

Figure 9.1 Interrupt vectors pointing to the *intmap* table.

A look at the code will clarify the details. The interrupt dispatch table *intmap* is defined in file *io.h*.

```
/* io.h -  fgetc, fputc, getchar, isbaddev, putchar */

#define INTVECI inint           /* input interrupt dispatch routine     */
#define INTVECO outint          /* output interrupt dispatch routine    */
extern  int     INTVECI();
extern  int     INTVECO();

#define NMAPS   0x20            /* number of intmap entries             */
struct  intmap  {               /* device-to-interrupt routine mapping  */
        char    ivec;           /* interrupt number                     */
        char    callinst;       /* the call instruction                 */
        word    intcom;         /* common interrupt code                */
        word    oldisr_off;     /* old int. service routine offset      */
        word    oldisr_seg;     /* old int. service routine segment     */
        int     (*newisr)();    /* pointer to the new int. ser. routine */
        word    mdevno;         /* minor device number                  */
        word    iflag;          /* if nonzero, call the old isr         */
        };

/*
 * NOTE: The intmap structure takes a total of 7 words or 14 bytes
 * per record.
 */

extern  struct intmap far *sys_imp;     /* pointer to intmap table      */
extern  int     nmaps;          /* number of active intmap entries      */

#define isbaddev(f)     ( (f)<0 || (f)>=NDEVS )

/* In-line I/O procedures */

#define getchar()       getc(CONSOLE)
#define putchar(ch)     putc(CONSOLE,(ch))
#define fgetc(unit)     getc((unit))
#define fputc(unit,ch)  putc((unit),(ch))

extern  int     _doprnt();      /* output formatter                     */
```

Each entry in *intmap* corresponds to one device. The interrupt type (from whence the interrupt comes) is stored at system initialization in the *ivec* field. Following this are three bytes for the CALL instruction. The segment:offset address of the previous interrupt service routine installed in the vector is saved in the *oldisr* entry. This address and the interrupt type in *ivec* are used to restore the interrupt vector to its previous value upon termination of PC-Xinu.

The *newisr* entry is a pointer (CS offset) to the PC-Xinu interrupt service routine code which is called by *intcom* when the device interrupt occurs. When called, *newisr* is passed the minor device number *mdevno* from the *intmap* table.

The *intmap* table itself is defined in *intmap.asm*:

```
; intmap.asm -
; low-level interrupt transfer table and dispatcher

        include dos.asm

tblsize equ     20h                     ; define max size of intmap table
stksize equ     100h                    ; max size of system stack

        dseg

        public  _sys_imp

intstack        db      stksize dup (?) ; interrupt stack
topstack        label   byte
_sys_imp        dd      far ptr intmap

        endds

        pseg

        public  pcxflag, sssave, spsave

pcxflag         dw      1               ; zero when rescheduling disabled
spsave          dw      ?               ; saved stack pointer register
sssave          dw      ?               ; saved stack segment register

;---------------------------------------------------------------------------
; intmap  --  interrupt dispatch table
;---------------------------------------------------------------------------
intmap  label   byte

        rept    tblsize

        db      ?                       ; ivec   - interrupt vector number
        call    intcom
        dd      ?                       ; oldisr - old isr from bios (s:o)
        dw      -1                      ; newisr - new isr code address
        dw      ?                       ; mdevno - minor device number
        dw      ?                       ; iflag  - interrupt flag
```

```
        endm

        ASSUME DS:NOTHING

;-------------------------------------------------------------------------
; intcom  --  common interrupt dispatcher
;-------------------------------------------------------------------------
; This procedure is interrupt handling code that is common to all
; interrupt service routines.
intcom  proc    near
        push    bp
        mov     bp,sp
        push    ax                      ; push registers
        push    bx
        mov     bx,[bp+2]               ; get pointer to intmap data
        mov     ax,cs:[bx+8]            ; get interrupt flag
        or      al,al                   ; zero?
        je      short nobios            ; yes, skip the call to the BIOS
        pushf                           ; push flags to simulate interrupt
        call    cs:dword ptr[bx]        ; call BIOS ISR
        cli                             ; be sure interrupts are back off
nobios:
        push    cx                      ; save rest of registers
        push    dx
        push    si
        push    di
        push    ds
        push    es
        mov     cs:sssave,ss            ; save stack environment
        mov     cs:spsave,sp
        mov     cx,cs                   ; get code segment
; bp+6 points to code segment where interrupt occurred
        cmp     cx,[bp+6]               ; check if we own interrupt
        jnz     short newstack
; time to do our ISR, since the stack and data segments are OK
        push    cs:word ptr[bx+6]       ; pass minor dev. no.
        call    cs:word ptr[bx+4]       ; call C ISR (saves si, di)
        add     sp,2                    ; deallocate parm. (C convention)
        jmp     short popregs
newstack:
; now set up temporary stack in DGROUP and do our ISR
        mov     ax,DGROUP
        mov     ds,ax                   ; set ds to DGROUP
```

```
        ASSUME  DS:DGROUP
        mov     ss,ax                   ; set up temporary stack in DGROUP
        mov     sp,offset topstack
        xor     ax,ax                   ; clear pcxflag to prevent resched
        xchg    ax,cs:pcxflag
        push    ax                      ; save for later
        push    cs:word ptr[bx+6]       ; pass minor dev. no.
        call    cs:word ptr[bx+4]       ; call our routine (saves si, di)
        add     sp,2                    ; deallocate parm. (C convention)
        pop     cs:pcxflag              ; restore pcxflag
        mov     ss,cs:sssave            ; restore old stack
        mov     sp,cs:spsave
        ASSUME  DS:NOTHING
popregs:
        pop     es                      ; restore all registers
        pop     ds
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        pop     bp
        add     sp,2                    ; remove pointer to intmap area
        iret
intcom  endp

        endps

        end
```

The REPT directive in the definition of *intmap* is a convenient way to allocate a number of identical entries without having to code each one individually. In this case, *tblsize* is 20H, which means that there will be 20H entries in the *intmap* table.

The *intmap* table is in the code segment, whose data is generally inaccessible by C programs. Since *intmap* entries must be filled in by high-level C initialization routines, *intmap.asm* defines a public data segment variable *sys_imp* containing the segment:offset pointer to *intmap*. This variable is available to C programs as a far pointer into the *intmap* data. Remember that the underscore at the beginning of the *sys_imp* name allows C programs to access the variable.

9.2.2 Implementation of the Interrupt Dispatcher

Intcom begins by pushing registers BP, AX, and BX on the stack. Since the dispatcher itself is an interrupt service routine and these registers are used for interrupt dispatch activities, it is necessary that they are saved before they are used. They will be popped off the stack prior to returning to the interrupted process.

Recall that upon entry to the interrupt dispatcher, the CS offset to the device's *intmap* entry (actually, to the *oldisr* address) is on the stack as a result of the CALL instruction. The BP register, which references the stack frame, is used to retrieve this pointer and to save it in the BX register. This pointer will be used to access all the components of the *intmap* entry. Before calling the C interrupt service routine, the remaining registers DX, SI, DI, DS, and ES are pushed on the stack, and the current SS:SP stack environment is saved in the segment : offset pair *sssave:spsave*.

The *intstack* array in *intmap.asm* deserves special comment. Interrupts which occur during a BIOS call may use data and stack segments that differ from those used by PC-Xinu. For PC-Xinu to handle such interrupts, the C interrupt service routine must be called with stack and data segments that agree with the memory layout given in Chapters 2 and 8. If the interrupt came from outside the PC-Xinu environment, *intcom* must set up a local stack prior to calling the C interrupt service routine. After the local stack has been used, *intcom* restores the SS:SP stack environment from the saved variables *sssave : spsave*.

The code illustrates another feature of 8088 assembly language. Data references by default use the DS segment register, but some *intmap* data entries reside in the code segment. To reference the code segment variables *pcxflag*, *sssave*, and *spsave*, a *segment override* is used. In assembly language, this is expressed using the prefix ''CS:'' on the instruction data. Since C programs refer to variables in the data segment by default, code segment variables ordinarily are accessible only through assembly language.

9.2.3 Deferred Rescheduling

A routine is said to be *reentrant* if simultaneous calls from more than one process can be executing code in the routine at the same time. Reentrant code abounds in operating systems that support multiple processes (*resched* is one of many examples in PC-Xinu). Routines that are reentrant normally use local variables belonging to the stack space of the user process; reference to any global variables is usually controlled by bracketing access to them with calls to disable and enable interrupts.

BIOS services are not reentrant, and since BIOS services typically enable interrupts, they cannot be trusted by themselves to protect their global variables from access by multiple processes when interrupts occur. PC-Xinu uses a single routine, *resched*, to switch from one process to another, and the only way *resched* can be called during BIOS services is through an interrupt (for example, from the clock or keyboard). When a BIOS routine is executing, it sets up its own data (and perhaps stack) segment to access its global variables. As described in the previous section, *intcom* code sets up a local stack in the PC-Xinu environment to handle interrupts which occur during BIOS services. These interrupts must *not* reschedule, since rescheduling can result in another call to the BIOS routine.

The *pcxflag* variable in *intmap.asm* serves as a gate to defer rescheduling. When *pcxflag* is reset to 0, PC-Xinu rescheduling is disabled, thereby preventing context switching and avoiding reentrant access by multiple processes. When *pcxflag* is set to 1, normal rescheduling by PC-Xinu is enabled. The code in the interrupt dispatcher shows that *pcxflag* is reset to 0 when the local stack is used to service BIOS interrupts. This is insurance against rescheduling by interrupts during BIOS activities which could wreak havoc in non-reentrant routines.

9.2.4 To BIOS or Not to BIOS

An *intmap* table entry for a device contains the segment : offset address of the old interrupt service routine which was in the device vector prior to PC-Xinu initialization. In most cases, this interrupt service routine is called by the interrupt dispatcher just before calling the PC-Xinu service routine. For example, keyboard interrupts *must* be processed by the BIOS before they are serviced by PC-Xinu. There are interrupts, however, which should *not* be handled by the old ISR. An example is *Ctrl-Break*, the type 1BH interrupt, which is invoked upon receipt of the *Ctrl-Break* key sequence. If the old ISR were to handle this first, it could terminate PC-Xinu and return control to MS-DOS, without giving PC-Xinu a chance to restore the interrupt vectors from the *intmap* table, an important program termination activity.

For this reason, *intmap* entries have an *iflag* field which, when nonzero, causes the interrupt dispatcher to call the old ISR before proceeding to the PC-Xinu handler. The value of this flag for each device is determined by the PC-Xinu configuration, which is discussed in Chapter 11.

9.3 Process Control Of Deferred Rescheduling

Pcxflag is reset by the interrupt dispatcher to defer rescheduling when the local stack is being used for interrupt processing. But there are occasions when a process may wish to defer rescheduling while leaving interrupts enabled. As an example, positioning the cursor on the screen and displaying a character at the current cursor position are separate BIOS calls: If a context switch occurs between these calls, another process may move the cursor before the current process can display the character. To make these activities essentially indivisible, a process calls *xdisable* to defer rescheduling. Rescheduling is deferred until the process calls *xrestore*.

The macros *xdisable* and *xrestore* defined in *kernel.h* are analogous to the *disable* and *restore* macros, except that they save and restore the *pcxflag* value. They expand to calls to the two assembly language service routines *sys_xdisabl* and *sys_xrestor*, which are similar to *sys_disabl* and *sys_restor* discussed in Chapter 5.

```
; xeidi.asm - _sys_xdisabl, _sys_xrestor, _sys_pcxget, _sys_getstk

        include dos.asm         ; segment macros

        dseg
; null data segment
        endds

        pseg

        public  _sys_xdisabl, _sys_xrestor, _sys_pcxget, _sys_getstk
        extrn   pcxflag:word
        extrn   sssave:word
        extrn   spsave:word

;---------------------------------------------------------------------------
; _sys_xdisabl  --  return pcxflag & disable context switching
;---------------------------------------------------------------------------
; int sys_xdisabl()
_sys_xdisabl    proc    near
        pushf
        cli                     ; disable interrupts
        xor     ax,ax
        xchg    ax,cs:pcxflag
        popf
        ret
_sys_xdisabl    endp

;---------------------------------------------------------------------------
; _sys_xrestor  --  restore pcxflag
;---------------------------------------------------------------------------
; sys_xrestor(ps)
; int ps;
_sys_xrestor    proc    near
        push    bp
        mov     bp,sp           ; C calling convention
        pushf
        cli                     ; disable interrupts
        mov     ax,[bp+4];      ; get passed flags word
        mov     cs:pcxflag,ax   ; reset pcxflag to passed value
        popf
        pop     bp
        ret
_sys_xrestor    endp
```

```
;------------------------------------------------------------------------
; _sys_pcxget  --  get the current value of pcxflag
;------------------------------------------------------------------------
; int sys_pcxget()
_sys_pcxget     proc    near
        pushf
        cli                     ; disable interrupts
        mov     ax,cs:pcxflag
        popf
        ret
_sys_pcxget     endp

;------------------------------------------------------------------------
; sys_getstk  --  get the stack parameters for panic printing
;------------------------------------------------------------------------
; sys_getstk(sssp,spsp)
; int *sssp,*spsp
_sys_getstk     proc    near
        ASSUME DS:DGROUP
        push    bp
        mov     bp,sp
        mov     bx,[bp+4]
        mov     ax,cs:sssave
        mov     [bx],ax
        mov     bx,[bp+6]
        mov     ax,cs:spsave
        mov     [bx],ax
        pop     bp
        ret
_sys_getstk     endp

        endps

        end
```

If rescheduling is deferred and the current process calls *resched* (indirectly through *send* or *signal*, for example), *resched* returns harmlessly. However, if a process changes its state and calls *resched* (indirectly through *receive* or *wait*, for example) with rescheduling deferred, the system will be left in an impossible state. Consequently, deferred rescheduling should follow the simple rule:

A process executing with rescheduling deferred can only call procedures that leave the process in the current state.

BIOS calls cannot change the state of a process, so they are safe to call with rescheduling deferred.

9.4 The Rules For Interrupt Processing

Because interrupt routines examine and modify global data structures like I/O buffers, these routines must be designed to prevent other processes from interfering with them. Generally, noninterference is guaranteed by making the interrupt routines uninterruptable. For example, PC-Xinu disables interrupts by calling an assembly language routine to clear the interrupt flag in the *FLAGS* register. Interrupts remain disabled even if an interrupt routine calls other procedures. When the high-level interrupt procedure returns, control passes back to the interrupt dispatcher, which restores the *FLAGS* register and returns to the place at which processing was originally interrupted. Only after the dispatcher returns are interrupts enabled again.

Interrupt routines may also enable interrupts by calling *resched*, if it switches to a process that has interrupts enabled. The PC-Xinu clock interrupt routine, for example, calls *resched* directly, and the keyboard interrupt routine calls *send* to indicate that a character is available, indirectly calling *resched*. In each case, when the call reaches *resched*, it might allow a process to execute that had interrupts enabled. So, shared data structures must be left in a valid state before calling any routines that switch context. To sum up:

Interrupts will be disabled when the dispatcher calls a high-level interrupt routine; the high-level routine must be designed to keep further interrupts disabled until it completes changes to global data structures.

There are several other issues to consider when building interrupt routines. For one, interrupt routines cannot keep interrupts disabled too long. If they do, devices will fail to perform correctly. For example, if the processor does not accept a character from a serial port before another arrives, data will be lost. So, interrupt routines must be designed to enable further interrupts as quickly as possible.

Another constraint arises because interrupt code is executed by whichever process happens to be running when the interrupt occurs. In particular, the interrupt routines must be designed so they work correctly even if executed by the null process. Recall that *resched* blindly assumes at least one process remains ready to run, so the null process must always be *current* or *ready*. The most important consequence is:

Interrupt routines can only call procedures that leave the executing process in the current or ready states.

So, interrupt routines may use primitives like *send* or *signal*, but they may not use primitives like *wait*.

9.5 Rescheduling While Processing An Interrupt

Should interrupt routines be allowed to reschedule? We already stated that interrupt routines cannot explicitly enable further interrupts while processing an interrupt, and that they must leave global data structures in a valid state before rescheduling. It might seem that they should not be allowed to reschedule either, because switching to a process that had interrupts enabled would start a sequence of interrupts piling up until the stack overflowed. Rescheduling is important, however, because it provides the only way that interrupt routines can affect the running process. We must convince ourselves that rescheduling from an interrupt is safe as long as global data structures are valid.

To understand why rescheduling is safe, consider the series of events leading to a call of *resched* from an interrupt handler. Suppose a process *P* was running with interrupts enabled when the interrupt occurred. The hardware uses *P*'s stack to save the registers and continues to let process *P* run the interrupt dispatcher. Because the interrupt mechanism disables interrupts, *P* executes the dispatch routine with interrupts disabled. Interrupts remained disabled when the dispatcher calls the high-level interrupt routine. Suppose now that the high-level interrupt routine calls *resched* which switches to another process, say *Q*. If *Q* happens to enable interrupts (e.g., by returning from a system call), another interrupt may occur. What prevents an infinite loop where unfinished interrupts pile up until the run-time stack overflows with interrupt procedure calls? Recall that each process has its own stack. Process *P* had one interrupt on its stack when it was stopped by the context switch. The new interrupt occurs while the processor is using *Q*'s stack. Before another interrupt can pile up on *P*'s stack, it must regain control of the CPU and enable interrupts. But *P* was running with interrupts disabled when it called the scheduler and context switch. The context switch saved *P* with interrupts disabled, so when it eventually switches back to *P*, it will restore the *FLAGS* register, and *P* will continue execution with interrupts disabled.

Interrupts remain disabled as *resched* returns to the high level interrupt routine and as the high-level interrupt routine returns to the interrupt dispatcher. Interrupts only become enabled again when the dispatcher returns to the spot at which the original interrupt occurred. So, interrupts cannot occur while process *P* is executing interrupt code (even though they can occur to another process while *P* is not executing). Only a finite number of processes exist at any time and each, in turn, can be processing at most one interrupt. Looking at this a different way, we can say that:

> *Rescheduling during interrupt processing is safe provided that (1) interrupt routines leave global data in a valid state before rescheduling, and (2) no procedure enables interrupts unless it disabled them.*

If you recall, you will see that we have used this rule in all the procedures built so far: a procedure that disables interrupts upon entry always restores them before returning to its caller; no routine ever enables interrupts explicitly. Because interrupts are disabled upon entry to the interrupt dispatcher, they are restored when it returns. The only exception to our rule about disabling and restoring interrupts is found in the initialization procedure which enables interrupts at system startup.

9.6 PC Interrupt Vectors

The PC interrupt vectors used by PC-Xinu can be classified into two categories: exception vectors and device interrupt vectors. Exception vectors are discussed in more detail in Chapter 19. For the moment, think of them as vectors corresponding to interrupts generated by the processor rather than by input/output devices. The exception and device interrupt vector addresses are collected together in the file *bios.h*.

```
/* bios.h */

/*------------------------------------------------------------------------
 *  ROM BIOS interface information for PCs
 *------------------------------------------------------------------------
 */

#define DB0VEC          0x00            /* divide by zero exception vec */
#define SSTEPVEC        0x01            /* single step exception vector */
#define BKPTVEC         0x03            /* breakpoint exception vector  */
#define OFLOWVEC        0x04            /* overflow exception vector    */

#define CLKVEC  0x08                    /* Clock interrupt vector       */
#define KBDVEC  0x09                    /* Keyboard interrupt vector    */
#define COM1VEC 0x0b                    /* COM1 interrupt vector        */
#define COM2VEC 0x0c                    /* COM2 interrupt vector        */
#define FLOPVEC 0x0e                    /* Floppy interrupt vector      */
#define PRLLVEC 0x0f                    /* Parallel port interrupt vec  */
#define CBRKVEC 0x1b                    /* Ctrl-Break interrupt vector  */

#define BIOSFLG 0x100                   /* BIOS flag for intmap         */
```

```
extern int _panic();                    /* exception handler           */
extern int cbrkint();                   /* ctrl-break handler          */
extern int clkint();                    /* clock interrupt handler     */
```

Of the seven device-related interrupt vectors listed, only *CLKVEC*, *KBDVEC*, and *CBRKVEC* are used by PC-Xinu.

9.7 Summary

High-level languages like C cannot always manipulate the machine registers or execute special instructions to handle interrupts. To avoid writing all interrupt code in assembly language, we have divided the work into two parts. A small assembly language dispatcher permits the bulk of interrupt handling to be written in a high-level language. The dispatcher fields interrupts, saves machine registers, and passes control to an appropriate high-level interrupt handler based on entries in a dispatch table. When the high-level handler returns, control passes back to the dispatcher which reloads registers and executes special instructions that restore the state and return to the interrupted program.

A few rules simplify the design of interrupt handlers. First, the interrupt handler must not enable interrupts explicitly; it may, however, reschedule to allow other processes to execute. (Of course, the interrupt routine must insure that global data structures are in a valid state before rescheduling). Second, because an interrupt routine may be executed by the null process, it must never call a procedure that will move the calling process out of the *current* or *ready* states. Third, interrupt routines must not leave interrupts disabled too long, or devices will fail to operate correctly. The length of time an interrupt can be delayed depends on the device hardware; a serial port, for example, need only be serviced before another character arrives.

FOR FURTHER STUDY

Tanenbaum [1976] and Stone [1972] describe interrupt mechanisms on various machines. Madnick and Donovan [1974] consider the details on an IBM 360 architecture. More information on the details of the 8088 can be found in the vendor's manual *iAPX 88 BOOK*.

Information on interrupt processing in general can be found in Watson [1970]. Lister [1979] describes dispatching.

EXERCISES

9.1 Rewrite the I/O interrupt dispatcher, minimizing the number of instructions executed by building a different copy of the dispatcher for each device. How many instructions can you save per interrupt?

9.2 What might happen if the interrupt dispatcher did not run with interrupts disabled?

9.3 Experiment with PC-Xinu by enabling interrupts upon entry to the interrupt dispatcher. See what happens if you disable PC-Xinu clock interrupt service. (See the next chapter for information on clock interrupts). Are you surprised at how long the system runs before crashing? Determine *exactly* why it crashes.

9.4 Suppose the hardware automatically switched context to a process that handled interrupts whenever one occurred. Would the system be easier or more difficult to design? Would that process be permitted to reschedule?

9.5 Hardware with a separate interrupt vector for each device works best if the software has one interrupt routine per device. Why does PC-Xinu use a central interrupt dispatcher? Modify the code to use device-specific interrupt service routines.

9.6 Calculate how many milliseconds can be spent per interrupt, assuming four devices, each receiving characters at 19.2 Kbaud (19.2 thousand bits per second, or approximately 1920 characters per second). Roughly, how many 8088 instructions can be executed per interrupt at this rate?