Interrupts

This lecture covers the use of interrupts and describes the vectored interrupt mechanism used on the IBM PC using the Intel 8259 Programmable Interrupt Controller (PIC).
After this lecture you should be able to: (1) choose between polling and interrupts to service a peripheral and justify your choice, (2) describe how the 8259 PIC handles multiple interrupt sources, and (3) write an ISR in 8088 assembly language to service interrupts generated by the 8259 PIC.

Introduction

The two common ways of servicing devices are by polling and by using interrupts. *Polling* means the status of the peripheral is checked periodically (typically by testing a "data register empty" bit in a status register) to determine whether it needs to be serviced, for example whether the device has data ready to be read.

The alternative is to use *interrupts*: a signal from the peripheral connects to the CPU's interrupt request input. The peripheral interface is designed to assert this interrupt request signal when it requires service (e.g. when it has data available or when it can accept more data). The result of asserting the interrupt signal is to temporarily suspend the currently-executing program and to cause an interrupt service routine (ISR) to be executed. The ISR transfers data to/from the peripheral.

In this lecture we cover the design of interrupt-driven I/O devices on IBM PC compatible architectures.

Choosing Between Polling and Interrupts

I/O devices such as printers, keyboards, etc. require that the CPU execute some code to "service" the device every once in a while. For example, incoming characters or keystrokes have to be read from a data register on the peripheral interface and stored in a buffer so that the application can retrieve them when they are needed.

Polling must be done sufficiently fast that data is not lost. For example, if a serial interface can receive up to 1000 characters per second and can only store the last character received, it must be checked at least once per millisecond to avoid losing data. Since we need to periodically check each device, regardless of whether it requires service or not, polling causes a fixed overhead for each installed device.

Another, possibly greater, disadvantage of polling is that polling routines must be integrated into each and every program that will use that peripheral. Programs must be written to periodically poll and service all the peripherals they use. Such tight coupling between the application and the hardware is usually undesirable except in the simplest embedded processor control systems.

On the other hand, an ISR is only executed when a device requires attention (e.g. a character has been received). Thus there is no fixed overhead for using interrupt-driven devices. In addition, since ISRs operate asynchronously with the execution of other programs, it is not necessary for application programs to worry about the details of the I/O devices.

However, responding to an interrupt typically requires executing additional clock cycles to save the processor state, fetch the interrupt number and the corresponding interrupt vector, branch to the ISR and later restore the processor state. In addition, ISRs are much more difficult to write and debug than other code because many types of ISR errors will "crash" the system.

Some factors to consider when deciding whether to use polling or interrupts include:

- Can the device generate interrupts? If the peripheral is very simple then it may not have been designed to generate interrupts. Very simple microcontrollers may not have interrupt hardware.

- How complex is the application software? If the application is a complex program that would be difficult to modify in order to add periodic polls

of the hardware then you may have to use interrupts. On the other hand, if the application is a controller that simply monitors some sensors and controls some actuators then polling may be the best approach.

- What is the maximum time allowed between polls? If the device needs to be serviced with very little delay then it may not be practical to use polling.
- What fraction of polls are expected to result in data transfer? If the rate at which the device is polled is much higher than the average transfer rate then a large fraction of polls will be "wasted" and using interrupts will reduce this polling overhead.

In general, you should use interrupts when the overhead due to polling would consume a large percentage of the CPU time or would complicate the design of the software.

Exercise 48: You are designing a simple furnace controller. It uses a microcontroller to read a temperature sensor and turn a heater on and off in response to the temperature. Would you use interrupts for this application?

Exercise 49: You are designing the driver software for a LAN interface card that will work in a general-purpose computer. Is this driver likely to make use of interrupts?

Exercise 50: Data is arriving on a serial interface at 4000 characters per second. If this device is serviced by polling, and each character must be read before another one is received, what is the maximum time allowed between polls? If each poll requires 10 microseconds to complete, what fraction of the CPU time is always being used up even when the serial port is idle? What if there were 8 similar devices installed in the computer?

Exercise 51: Data is being read from a tape drive interface at 100,000 characters per second. The overhead to service an interrupt and return control to the interrupted program is 20 microseconds. Can this device use an ISR to transfer each character?

Alternative Approaches

It's also possible to use a mixture of interrupt and polled devices. For example, a device can be polled by an ISR that executes periodically due to a clock interrupt. This removes the need to include polling routines in each application without needing to add interrupt request hardware to the peripheral.

We can also poll several different devices from a common ISR. This may actually be more efficient that having each device issue independent interrupts.

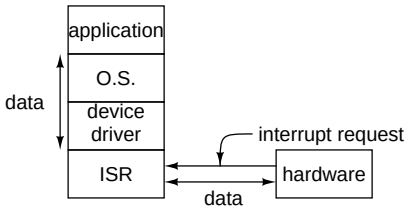
It is also common for devices to buffer multiple bytes and issue an interrupt only when the buffer is full (or empty). The ISR can then transfer the complete buffer without incurring the interrupt overhead for each byte. For example, modern PC serial interfaces can store up to 16 bytes before issuing an interrupt. This cuts down the interrupt overhead by up to 16.

Because interrupts occur due to events outside the computer's control, it is usually difficult to predict the exact sequence and rate in which interrupts will happen. In applications where loss of data cannot be tolerated (e.g. where safety is a concern) the designer must ensure that all of the devices serviced by interrupts can be properly serviced under the worst-case conditions. Typically this involves a sequence of nested interrupts happening closely one after another in a particular order. In some of these systems it may be better to use polling rather than interrupts in order to ensure correct worst-case behaviour.

Exercise 52: Consider a monitoring system in a nuclear power plant. The system is hooked up to hundreds of sensors, each of which can indicate an error condition. It is difficult to predict exactly how often and in what order these error conditions will happen. Would you design the system so that alarm conditions generated interrupts? Why or why not?

Operating Systems, Device Drivers and ISRs

General-purpose computers (as well as many embedded systems) use operating systems to provide device-independent I/O. The operating system converts a generic I/O request (e.g. "write this buffer to the standard output") into the low-level `IN` and `OUT` instructions to control a specific piece of hardware (e.g. a disk drive) and transfer data.



The software that carries out the device-specific control and I/O operations is called a *device driver*. Typically a device driver is divided into a "slow" part that is called by the operating system (the "kernel") and a "fast" part that is invoked by an interrupt (i.e. it is an ISR). These two software routines communicate using shared data structures.

The actual details vary widely depending on the operating system and the hardware.

Maskable, Non-Maskable and Software Interrupts

Like many other processors, the 80386 has two types of interrupts: maskable and non-maskable. Maskable interrupts (asserted on the INTR pin) can be disabled by clearing the interrupt-enable flag (IF bit) in the flags register using the `CLI` instruction.

Non-maskable interrupts (asserted on the NMI pin) cannot be disabled. Thus NMI is usually used for very high priority events such as imminent loss of power or a hardware fault. For example, on the PC NMI is asserted if the hardware discovers a memory error.

Software interrupts cause the same interrupt processing as maskable and non-maskable interrupts but they are caused by executing an `INT` instruction.

In addition, certain error conditions (such as divide by zero) can cause *exceptions* which behave in the same way as software interrupts.

Interrupt Processing

A maskable interrupt causes an interrupt acknowledge cycle (similar to a read cycle) which reads a 1-byte interrupt type from the interrupting peripheral. The interrupt type (which is not the same as the interrupt "number") is then multiplied by four (4) and an interrupt vector is fetched from this address.

An NMI always uses the interrupt vector for interrupt type 2, thus allowing it skip this interrupt acknowledge cycle and thus execute faster.

For a software interrupt the interrupt type is supplied in the instruction and so, again, no interrupt acknowledge cycle is required.

The following sequence of events happens in response to an interrupt:

1. the current instruction is completed. If a lengthy instruction, such as a divide, is in progress this could take tens of bus cycles
2. an interrupt acknowledge cycle is run[1] and the CPU reads an interrupt type from a special peripheral that recognizes the interrupt acknowledge cycle and responds with the interrupt type
3. the CPU saves the processor context (flags, IP and CS registers are pushed on the current stack) to preserve the values of the processor registers ("processor context") when the interrupt happened
4. the interrupt-enable flag (IF) is cleared, thus preventing the interrupt request signal from immediately causing another interrupt
5. an interrupt vector (the address of the start of the ISR) is retrieved from memory by reading 4 bytes (offset and segment) from an address equal to the interrupt type multiplied by 4.
6. the CPU branches to the address that was retrieved in the previous step and this begins execution of the ISR

The first two steps are skipped in the case of NMI and software interrupts.

Exercise 53: How could you invoke the NMI handler on a PC if you had to regain control after it executed?

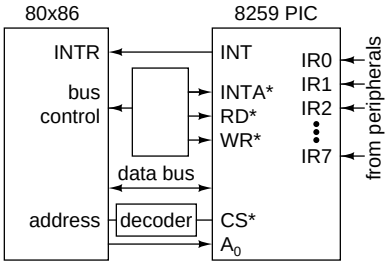
Exercise 54: What memory locations store the NMI interrupt vector? The ISR for NMI is located at address E000:0FBE (segment:offset). Draw a diagram showing the contents of each byte of these memory locations.

Exercise 55: In "real mode" each 80x86 interrupt vector requires 4 bytes. What is the maximum number of bytes used up by an interrupt vector table?

[1] For obscure reasons the CPU actually performs two interrupt acknowledge bus cycles separated by a number of idle bus cycles, but we will consider it as a single bus cycle.

The 8259 in the IBM PC Architecture

The 80x86 CPUs only have one interrupt request pin. Although simple systems may only have one interrupt source, most systems must have some way of dealing with multiple interrupt sources. The Intel "way of doing things" is to use a chip called a programmable interrupt controller (PIC). This chip takes as inputs interrupt request signals from up to 8 peripherals and supplies a single INTR signal to the CPU as shown below:



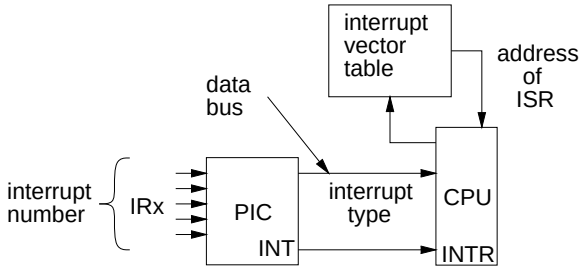
The PIC has 3 purposes:

1. It allows the individual interrupts to be enabled or disabled (masked).

2. It prioritizes interrupts so that if multiple interrupts need to be serviced at the same time the one with the highest priority is serviced first. The priorities of the interrupts are fixed, with input IR0 having the highest priority and IR7 the lowest. Interrupts of a lower priority not handled while an ISR for a higher-level interrupt is active.

3. It outputs the interrupt type that the CPU reads during the interrupt acknowledge cycle. This tells the CPU which of the 8 possible interrupts occurred. The PIC on the IBM PC is programmed to respond with an interrupt type of 8 plus the particular interrupt signal (e.g. if IR3 was asserted the CPU would read the value 11 from the PIC during the interrupt acknowledge cycle).

The following diagram shows how each of the interrupt request lines to the PIC can potentially cause an interrupt request to be made to the CPU. The CPU reads the interrupt type from the PIC during the interrupt acknowledge cycle and then uses this type to look up the address of the ISR in the interrupt vector table.



On the IBM AT and later models there are more than 8 interrupt sources and there are two PICs. The slave PIC supports an additional 8 interrupt inputs and requests an interrupt from the master PIC as if it were an interrupting peripheral on IR2.

Exercise 56: What is the maximum number of interrupt sources that could be handled using one master and multiple slave PICs?

Exercise 57: Compare this approach to that used for vectored interrupts on typical 68000 systems. How many interrupt request lines are there? Are they active-high or active-low? How many interrupt sources can be connected directly to a 68000? What if a wired-or configuration is used? What if a priority encoder is used? What device supplies the interrupt number or interrupt vector in a typical 68000 system?

Interrupt Number and Interrupt Type

A common source of confusion is the difference between the interrupt number, which is the interrupt request pin on the PIC that is asserted by a peripheral and the interrupt type which is the value read by the CPU during the interrupt acknowledge cycle or supplied in an `INT` instruction.

The interrupt inputs to the PIC are connected as follows on a IBM PC-compatible system:

interrupt number	interrupt type	device	
0	8	timer	highest
1	9	keyboard	
2	10	reserved	
3	11	serial port 2	
4	12	serial port 1	
5	13	hard disk	
6	14	floppy disk	
7	15	printer 1	lowest

The following are some of the other interrupt types that are pre-defined on 80x86 CPUs:

interrupt type	cause
0	Divide by Zero
1	Single Step
2	NMI
3	Breakpoint
4	Overflow
.	.
8 to 255	implementation-dependent

Note that these are not the same as the interrupt numbers.

Exercise 58: On an IBM PC-compatible system what interrupt number is used for a floppy-disk interrupt? What interrupt type will the CPU see for this interrupt? At what addresses will the CPU find the interrupt vector for this interrupt?

Exercise 59: When the a key on the keyboard is pressed, which input on the 8259 PIC will be asserted? What will the signal level be? What value will the 80386 read from the PIC during the interrupt acknowledge cycle?

Programming the 8259 Interrupt Controller

The initialization of the PIC is rather complicated because it has many possible operating modes. The PIC's operating mode is normally initialized by the BIOS when the system is booted. We will only consider the standard PIC operating modes used on the IBM PC and only a system with a single (master) PIC.

In it's standard mode the PIC responds to an interrupt request as follows:

- if the PIC believes that no ISR for the same or a higher level is active, the interrupt request (INTR) signal to the CPU is asserted
- if the CPU's interrupt enable flag is set then an interrupt acknowledge cycle will happen when the current instruction terminates
- during the interrupt acknowledge cycle the highest-priority interrupt request is captured and saved ("latched") in the PIC's interrupt request register (IRR) and then the interrupt type for this interrupt is read by the CPU from the PIC. An interrupt acknowledge actually takes two clock cycles.

  The CPU uses the interrupt type to look up the address of the ISR and runs it
- at the end of the ISR a command byte (20H) must be written to the PIC register at address 20H to re-enable interrupts at that level again. This is called the 'EOI' (end-of interrupt) command.

Exercise 60: Why does the ISR have to issue an EOI instruction? How does the PIC know which ISR is terminating?

During normal operation only two operations need to be performed on the PIC:

1. Disabling (masking) and enabling interrupts from a particular source. This is done by reading the interrupt mask register (IMR) from location 21H, using an AND or OR instruction to set/clear particular interrupt mask bits.
2. Re-enabling interrupts for a particular level when the ISR for that level complete. This is done with the EOI command as described above.

Masking/Enabling Interrupts

There are three places where interrupts can be disabled: (1) the PIC interrupt mask, (2) the PIC priority logic, and (3) the CPU's interrupt enable flag.

First, if the PIC interrupt mask bit is set then the interrupt request will not be recognized. Second, if the PIC believes an ISR for a higher level interrupt is still executing due to no EOI command having been

given for that interrupt level it will not pass on interrupts of the same or lower levels. Finally, if the interrupt enable bit in the CPU's flags register is not set then the interrupt request signal from the PIC will be ignored.

Exercise 61: How do an interrupt "mask" bit (e.g. in the PIC) and an interrupt "enable" bit (e.g. in the CPU flags register) differ?

Note that the CPU's interrupt enable flag is cleared when an interrupt happens and is restored when the process returns from the ISR via the IRET instruction. This means that ISRs can't be interrupted (not even by a higher-level interrupt) unless interrupts are explicitly re-enabled in the ISR.

Exercise 62: Can interrupts on an IBM-PC compatible computer be nested (i.e. can an ISR be interrupted)? If so, under what conditions? What instruction(s) are required to do this?

Exercise 63: How many levels deep can interrupts be nested on the IBM PC if the ISR does not re-enable interrupts? If it re-enables interrupts but does not issue EOI to the PIC? If it does both? In each of these cases how much space would be required on the interrupted program's stack to hold the values pushed during the interrupt acknowledge cycle?

Interrupt Latency

Often a peripheral must be serviced within a certain time limit after an event. For example, a character must be read from an input port before the next one arrives.

The interrupt *latency* is the maximum time taken to respond to an interrupt request. This will include the time it takes for the current instruction to complete plus the time for the CPU to respond to the interrupt (e.g. save the CS, IP and flag registers on the stack, acknowledge the interrupt and fetch the interrupt vector). If an ISR is already executing and cannot be interrupted then this also increases the interrupt latency.

Interrupt routines should be kept as short as possible to minimize the interrupt latency. Typically this involves having the ISR store values in a buffer or set flags and then having the bulk of the processing performed outside the ISR. A typical "device driver" consists of an ISR that executes only time-critical functions such as reading/writing data from/to the peripheral and another portion that deals with higher-level issues such as moving the disk drive head, checking for errors, etc.

Race Conditions, Critical Sections and Deadlock

A *race condition* is unpredictable behavior that depends on the timing of events. Here we are concerned with race conditions that arise because ISRs execute asynchronously with respect to other code.

Consider the following sequence of code that decrements the variable `count` (which, for example, could represent the number of bytes stored in a buffer):

```
 mov     ax,count
-----------------> ISR runs here
 sub     ax,1
 mov     count,ax
```

Consider what would happen if an ISR interrupted this code immediately after the first `mov` instruction and proceeded to increment `count`. When the ISR returns, the code will save the *old* value of count (now in AX) minus one to count and thus nullify the increment operation performed by the ISR.

Exercise 64: Assume count is initially set to 5. What is the value of count after the ISR executes? What is the value after the above routine ends? What would have been the value if the ISR had executed before the first `mov` instruction?

A *critical section* is a part of a program that should not be interrupted (typically because doing so would introduce a race condition). To prevent interrupts while this code is executing, a CLI instruction is placed before the critical section and an STI instruction after it.

Race conditions are introduced whenever a data structure can be modified by both the ISR and non-ISR code. Accesses to such data *must* be placed in a critical section. An even better approach is to redesign data structures to eliminate such shared-write variables.

Deadlock happens when two threads of execution (e.g. ISR code and non-ISR code) prevent each other from continuing. An example might be an ISR that needs to be "enabled" before it passes data to a program. If, for some reason, the program decides to wait for data to become available from the ISR without first enabling it, the program will "deadlocked."

Edge- and Level-Triggered Interrupts

Interrupt request signals can be designed to be:

- edge-triggered: the interrupt acts as a clock and the rising (or falling) edge of the interrupt signal causes an interrupt to be recorded), or
- level-triggered: the interrupt controller samples the interrupt signal at certain times and records an interrupt if the input is asserted at that time.

On many microprocessor systems the interrupt request outputs from multiple peripherals can be connected in a wired-or configuration to one active-low interrupt request input.

However, on the PC both INT and IRx are active-high signals and thus cannot be directly connected in a wire-or'ed configuration. In addition, the 8259 PIC is configured for edge-triggered interrupt inputs.

Exercise 65: Is it possible for several devices to share the same PIC interrupt request line? What would happen if one device requested an interrupt while another's interrupt was still pending?

Sample 80x86/8259 ISR

The code below shows an 80x86 assembly language program that includes an ISR. The program sets up an ISR for interrupt type 8 (interrupt number 0, the timer interrupt on the IBM PC). The ISR simply decrements a count. The main program waits until the count reaches zero and then terminates.

The timer interrupt on the IBM PC is driven by a clock that generates one interrupt every 55 milliseconds. With the initial count value provided below the program waits for 15 seconds before terminating.

The main program saves and restores the previous timer interrupt vector.

When the ISR begins execution only the IP and CS registers will have been initialized. Any other segment registers that will be used in the ISR must be explicitly loaded. In this case (a DOS .com file) the code and data segments have the same segment register values so DS can be loaded from CS.

On entry to the ISR only the IP, CS and flags registers will have been saved on the caller's stack.

All other registers modified by the ISR must be saved when starting the ISR and restored before returning. Otherwise the state of the interrupted code will be changed by the ISR and this is likely to cause seemingly-random failures in other programs.

The code below uses segment over-rides: the segment register to be used to form the 20-bit address is explicitly given along with the offset.

```
;
; example of program using an ISR for
; IBM PC timer interrupt
;

isrvec  equ     4*(8+0) ; location of vector for IR0

                        ; there are 4 bytes/vector and
                        ; PIC supplies x+8 for IRx

code segment public     ; .COM file setup
     assume  cs:code,ds:code
     org     100h

start:
        mov     ax,0    ; use ExtraSegment to access
        mov     es,ax   ; vectors in segment 0

; save old interrupt vector

        mov     ax,es:[isrvec]
        mov     prevoff,ax
        mov     ax,es:[isrvec+2]
        mov     prevseg,ax

; set up new vector

        cli     ; disable interrupts until
                ; vector update is complete

        mov     ax,offset isr
        mov     es:[isrvec],ax
        mov     ax,cs
        mov     es:[isrvec+2],ax

        sti     ; re-enable interrupts

; wait until ISR decrements count to zero

loop:   mov     ax,count
        cmp     ax,0
        jnz     loop

; restore old interrupt vector

        cli     ; disable interrupts until
                ; vector update is complete

        mov     ax,prevoff      ; restore prev.
        mov     es:[isrvec],ax  ; offset/segment
        mov     ax,prevseg
        mov     es:[isrvec+2],ax

        sti                     ; re-enable
                                ; interrupts
```

```
; return to DOS

        int     20h

; storage for demonstration program

count   dw      273
prevoff dw      ?
prevseg dw      ?

; The ISR:

isr:
        mov     cs:tmpax,ax      ; save working registers
        mov     ax,ds
        mov     cs:tmpds,ax

        mov     ax,cs   ; set up DS
        mov     ds,ax

        mov     ax,count
        cmp     ax,0    ; don't decrement if already zero
        jz      isr1
        sub     ax,1    ; decrement count
        mov     count,ax
isr1:

        mov     al,20h  ; write EOI command to 8259 PIC
        out     20h,al  ; to re-enable interrupts

        mov     ax,tmpds      ; restore working registers
        mov     ds,ax
        mov     ax,cs:tmpax

        iret            ; return from ISR and
                        ; re-enable interrupts

tmpax   dw      ?
tmpds   dw      ?

code    ends
        end     start
```

Exercise 66: Why must interrupts be disabled while updating the interrupt vector?

Exercise 67: How will the PC's time of day change when this program is run? What would happen if the interrupt were not restored?

Exercise 68: Could "the stack" be used to save the values of the registers that will be changed in the ISR? Which stack will be used? What are the advantages and disadvantages of doing so?