PC Interrupt Structure and 8259 DMA Controllers

This lecture covers the use of interrupts and 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: decide and explain why interrupts should (or should not) be used to service a particular peripheral and describe how the 8259 PIC handles multiple interrupt sources.

1 Review of Interrupts

Many peripheral devices such as serial interfaces, keyboards and real-time clocks need to be serviced periodically. For example, incoming characters or keystrokes have to be read from the peripheral or the current time value needs to be updated from a periodic clock source.

The two common ways of servicing devices are by polling and by using interrupts. *Polling* means that a status bit on the interface is periodically checked to see whether some additional operation needs to be performed, for example whether the device has data ready to be read. A peripheral interface can also be designed to assert an *interrupt* input to the CPU when it requires service. The result of asserting the interrupt signal is to interrupt normal flow of control and causes an interrupt service routine (ISR) to be executed to service the device.

Polling must be done sufficiently fast that data is not lost. Since each poll requires a certain number of operations, this creates a certain minimum overhead (fraction of available CPU cycles) for servicing each device. In addition, these polling routines must be integrated into each program that executes on the processor.

On the other hand, since an ISR is only executed when an interrupt occurs, there is no fixed overhead for servicing interrupt-driven devices. However, responding to an interrupt requires some additional overhead to save the processor state, fetch the interrupt number and then the corresponding interrupt vector, branch to the ISR and later restore the processor state.

In general, it is advantageous to use interrupts when the overhead required by polling would consume a large percentage of the CPU time or would complicate the design of the software. It is advantageous to use polling when the overhead of servicing an interrupt is a large percentage of the time available to service the device.

Exercise: 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: 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?

It is also possible to use a mixture of interrupt and polled devices. For example, devices can be polled by an ISR that executes periodically due to a clock interrupt. 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 buffer without an ISR overhead for each byte.

Because interrupts arrive under control of multiple events outside the computer's control, it is usually difficult to predict the exact sequence in which interrupts will happen. In applications where loss of data cannot be tolerated (e.g. where safety would be affected) the designer must ensure that all of the devices serviced by interrupts can be properly serviced under 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 easier to use polling rather to help ensure correct worst-case behaviour.

Exercise: Responding to an interrupt typically takes considerably longer than polling a status bit. Why are interrupts useful?

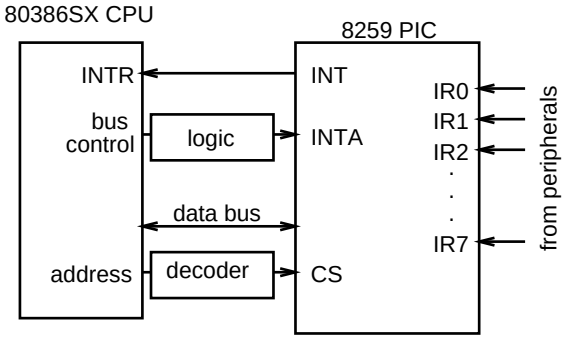
2 Maskable and Non-Maskable Interrupts

Like most other processors, the 80386 has two types of interrupts: maskable and non-maskable. Maskable interrupts (the INTR pin) can be disabled by clearing the IF bit (flag) in the processor status word (a special register used to control the operation of the CPU). Non-maskable interrupts (NMI pin) cannot be disabled. An maskable interrupt causes an interrupt acknowledge cycle (similar to a read cycle) which is used to read a 1-byte interrupt type from the interrupting peripheral. The interrupt type (which is not the same as the interrupt "number") is then used to fetch an interrupt vector which is stored in a table in memory. A NMI always uses the interrupt vector for interrupt type 2, thus allowing it execute faster.

Exercise: In "protected mode" each 386 interrupt vector requires 8 bytes. What is the maximum number of bytes are used up by an interrupt vector table?

3 The 8259 in the IBM PC Architecture

The 80386 CPU only has one interrupt request pin. Although simple systems may only have one interrupt source, more complex 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 each of the individual interrupts to be enabled or disabled (masked).
2. It prioritizes interrupts so that if multiple interrupts happen simultaneously 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 lower priority not handled while an ISR for a higher-level interrupt is active.
3. It provides an 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 PIC has two control registers that can be read or written. On the IBM PC the address decoder for PIC places these two registers in the I/O address space at locations 20H and 21H.

Unlike many other microprocessors both INT and IRx are active-high signals and on the IBM PC the IRx inputs are configured to be edge-triggered.

The interrupt inputs to the PIC are connected as follows:

interrupt	device
0	timer
1	keyboard
2	reserved
3	serial port 2
4	serial port 1
5	hard disk
6	floppy disk
7	printer 1

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

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

4 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 used on the IBM PC and only a system with a single (master) PIC.

In it's standard mode the PIC operates as follows:

- If a particular interrupt source is not masked then a rising edge on that interrupt request line is captured and stored ("latched"). Multiple interrupt requests can be "pending" at a given time.
- if 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 and the interrupt type for the highest pending interrupt is supplied by the PIC to the CPU
- 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.

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.

Exercise: What is the difference between an interrupt "mask" bit and an interrupt "enable" bit?

If the PIC interrupt mask bit is set then the interrupt request will not be recognized (or latched). If the PIC believes an ISR for an higher level interrupt is still executing due to no EOI command having been given for that interrupt level it will not allow interrupts of the same or lower levels. If the interrupt enable bit in the CPU's PSW is not set then the interrupt request signal from the PIC will be ignored.

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.

Interrupt routines should be kept as short as possible to minimize the interrupt latency (see below). 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.

It's possible to allow the CPU to interrupt an ISR (resulting in *nested interrupts*) by setting the interrupt enable bit with the STI instruction.

Exercise: 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 if 8 bytes are saved during each interrupt?

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 on 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 in-

terrupt (e.g. save the CS, EIP 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.

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.

Exercise: The 8259 PIC is configured for edge-triggered interrupts. Is it possible to share the interrupt request inputs by wire-OR'ing several interrupt sources? Why or why not? What if the inputs were active-low?