Nano-Kernel : A Bare Metal OS

## Part 6 - Handling Device Interrupts

In the original PC, included an i8259 *programming interrupt chip (PIC)* could generate an interrupt request (IRQ) numbered 0...7. This was expanded to 16 IRQs in the 80286 by using two *i8259*s, and that number was the standard until the Pentium processors. More recent versions of the hardware replace the 16 IRQs with an *Advanced Peripheral Interface Controller APIC*, which allows up to 1024 device IRQs (among other things, both good and bad!). This section asks you to implement the interrupt handlers for the 16 device interrupts of the early PC.

### Implement the 16 IRQs

* Modify “intr.h” to include the definition for an INTR and INTR2 macro. The hardware does **not** push an error code, so these macros will be almost a direct copy of the EXCP and EXCP2 macros (not the ERR macros).
* Add an interrupt offset number to the intr.h file, make it initially 32, so that IRQ 0 corresponds to interrupt 32, and IRQ 1 to interrupt 33, etc.
* Modify the “intr.S” to invoke your INTR macro to construct the 16 hardware interrupts (using the interrupt number NOT the IRQ number)
* Modify the “handler.h” file to add the interrupt templates that your INTR2 macro will generate for each of the 16 interrupts
* Modify the “handler.c” file to create an interrupt handler for your device - for now, it can just print the interrupt that was detected. **There will be a big difference from the error handlers - interrupts are NOT errors!**
* Modify the “handler.c” file to register your 16 interrupt handlers

### Test your Interrupt Handler

Modify your “kernel” to execute an “INT $32” which would correspond to IRQ and verify that it prints the IRQ and then your kernel continues running.

### The i8259 Interrupt Controller

The 8259 was the interrupt controller chip for the 8086 system[[1]](#footnote-1). For many years, this was actually a discrete component on the motherboard and external to the CPU. The 8086 had one, and later 80286 through the 80486 had two. Later architectures have the chip integrated into the CPU’s architecture, but the newer chipsets are still backward compatible with the 1970’s i8259.

Each 8259 can handle 8 interrupts. For systems that have two, one is the master and the other the slave. The slave is *cascaded* into the master on interrupt request (IRQ) 2. Thus, all eight of the slave’s interrupts generate an IRQ 2 to the PC.

The 8259 prioritizes interrupts in ascending order, so that the highest priority has the least IRQ number. Thus, the order of interrupts is: 0,1,2 { 8, 9, 10, 11, 12, 13, 14, 15 }, 3, 4, 5, 6, 7. Thus, if there are three interrupts pending: IRQ 6, IRQ 9, and IRQ 15, the interrupts will be processed as: IRQ 9, IRQ 15, IRQ 6.

The 8259 is a *programmable interrupt controller* (PIC). The programming interface is through its I/O ports (using IN and OUT instructions) rather than memory mapped I/O. The I/O assignments are:

* **Master PIC (0-7)**  
  0x20 - Control (A0 = 0) , 0x21 - IRQ Masking (A0 = 1)
* **Slave PIC (8-15)**  
  0xA0 - Control (A0 = 0) , 0xA1 - IRQ Masking (A0 = 1)

The device must be initialized by writing a series of initialization command words (ICW). During operation, the status of the device is configured by writing a series of operational command words (OCW). A summary of the registers follows, see the 8259 datasheet for details.

**ICW1 - Initialization Command Word 1 80x86 value: 0x11**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | A7 | A6 | A5 | 1 | LTIM | ADJ | SNGL | IC4 |

**A7:A5** - Used in “8085” mode only, set to 0 on 8086 systems

**LTIM** - Level or edge triggered interrupts

* 1 - All interrupts are level triggered
* 0 - All interrupts are edge triggered

**ADI -** Address interval (8085 mode only)

**SNGL** - Single or Master/Slave

* 1 - Single PIC
* 0 - Cascaded PICs

**IC4 -** Initialization command word 4 present

* 1 - 4 command words will be sent
* 0 - 3 command words will be sent

**ICW2 - Initialization Command Word 2 80x86 value: 0x20**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 |

**A<15:8>** - In 8085 mode, high byte of ISR address, in 8086 mode 8-bit vector offset

Must be a multiple of 8! (16, 24, 32, 40, …). Our default is offset 32 (0x20)

**ICW3 - Initialization Command Word 3 80x86 value: 0x04 / 0x02**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | S7 | S6 | S5 | S4 | S3 | S2 | S1 | S0 |

Master mode: A set bit indicates that there is a slave device present on that IRQ (1 << 2)

Slave mode: Indicates the IRQ that the slave device will present to the master (0x02)

**ICW4 - Initialization Command Word 4 80x86 value: 0x01**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 0 | SFNM | BUF | M/S | AEOI | uPM |

SFNM - Special Fully Nested Mode

* 1 - Special Fully nested mode
* 0 - Not special fully nested mode

BUF - Buffered Mode

* 1 - Buffered Mode
* 0 - Non-Buffered Mode

M/S - Master Slave Buffered Mode

* If BUF = 1
  + 1 - Master mode
  + 0 - Slave mode
* If BUF = 0: no impact

AEOI - Automatic End-of-Interrupt

* 1 - Automatic end-of-interrupt
* 0 - Normal end-of-interrupt

uPM - Microprocessor Mode

* 1 - 8086 Mode
* 0 - MCS-85 Mode

**OCW1 - Operational Command Word 1**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |

**Mask register, indicates that IRQm is masked**

* 1 - IRQ is *masked* and ignored
* 0 - IRQ is *unmasked* and will be processed

**OCW2 - Operational Command Word 2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | R | SL | EOI | CMD | | L3 | L2 | L1 |

***End of Interrupt Command <RL : SL : EOI >***

|  |  |  |  |
| --- | --- | --- | --- |
| **R** | **SL** | **EOI** | **Action** |
| 0 | 0 | 1 | Non-specific EOI |
| 0 | 1 | 1 | Specific EOI |
| 1 | 0 | 1 | Rotate on Non-Specific |
| 1 | 0 | 0 | Rotate in Auto Mode (set) |
| 0 | 0 | 0 | Rotate in Auto Mode (clear) |
| 1 | 1 | 1 | Rotate on specific EOI |
| 1 | 1 | 0 | Set priority command |
| 0 | 1 | 0 | No operation |

**CMD -** Select OCW2 by setting CMD = 0b00.

**IRQ Level to Be Acted Upon <L3:L2:L1>**  - 3-bit IRQ number

**OCW3 - Operational Command Word 3**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | ESMM | SMM | CMD | | P | RR | RIS |

|  |  |  |
| --- | --- | --- |
| ESMM | SMM | Action |
| 0 | 0 | No action |
| 0 | 1 |
| 1 | 0 | Reset special mask |
| 1 | 1 | Set special mask |

**CMD -** Select OCW3 by writing 0b01 to bits <4:3>

**P** - Polling

* 1 - Poll command
* 0 - No Poll command

|  |  |  |
| --- | --- | --- |
| RR | RIS | Action |
| 0 | 0 | No action |
| 0 | 1 |
| 1 | 0 | Read IR on next nRD pulse |
| 1 | 1 | Read IS reg on next nRD pulse |

### 8259 Initialization

There are two different initialization processes - one for the master, and the other for the slave. Once started the 8259 will await either 3 or 4 initialization command words. The A0 bit indicates whether the command byte is sent to the control (0x20 / 0xA0) address, or else the masking byte (0x21 / 0xA1).

Initialize the Master:

1. Mask: **0xff** to **0x21** (Mask all interrupts)
2. ICW1: **0x11** to **0x20** (edge trigger, cascaded PIC, ICW4 required)
3. ICW2: **0x20** to **0x21** (IRQ0...7 maps to INT32...INT39)
4. ICW3: **0x04** to **0x21** (Slave connected on IRQ2)
5. ICW4: **0x01** to **0x21** (8086 mode)
6. OCW3: **0x68** to **0x20** (Clear specific mask)
7. OCW3: **0x0a** to **0x20** (Read IR on next read)

Initialize the Slave:

1. Mask: **0xff** to **0xA1** (Mask all interrupts)
2. ICW1: **0x11** to **0xA0** (edge trigger, cascaded PIC, ICW4 required)
3. ICW2: **0x28** to **0xA1** (IRQ0...7 maps to INT40...INT47)
4. ICW3: **0x02** to **0xA1** (Slave is IRQ2)
5. ICW4: **0x01** to **0xA1** (8086 mode)
6. OCW3: **0x68** to **0xA0** (Clear specific mask)
7. OCW3: **0x0a** to **0xA0** (Read IR on next read)
8. Mask: **0xfb** to **0x21** (Unmask at least interrupt 2 in master)

### 8259 Setting the Mask

Masking allows us to select which IRQs are able to generate interrupts. When an IRQ is masked it is ignored. This is good practice to prevent spurious interrupts (an actual problem) in early systems.

If the masks are split into low and high (8-bits each):

1. Mask: **low nibble** to **0x21**
2. Mask: **high nibble** to **0xA1**

We must ensure that the low nibble always has the slave 8259 enabled or else we will unintentionally disable it: low = low & ~(1 << 2)

### 8259 Ending the Interrupt

Just as the Intel CPU needs to have its interrupt flag cleared (CLI), the 8259 needs to know when we are done with the current interrupt. This is accomplished by writing a 0x20 to the command port of the 8259 that generated the interrupt.

Command 0x20 is 0b0010\_0000, which corresponds to: CMD = 00, so OCW2. OCW2 bit <7:5> = 001 - non-specific end-of-interrupt command, all other bits are 0, so no other effects.

If the interrupt originated in the slave 8259, then the slave and the master need to have the end-of-interrupt byte written.

### An 8259 Utility

The next step is to develop an 8259 utility for our bare-bones OS. It needs to: support device initialization, masking and unmasking interrupts, and handling the end-of-interrupt commands. Code outside the utility should not need to know \*anything\* about the low-level operations - it should just have some functions to call that create the illusion of a highly capable device. To that end, we need to develop some C code to do the following:

* Initialize the device driver, including any “global state” and initialize the 8259s (master and slave)
* Set the enabled interrupts
* Get the enabled interrupts
* Clear the end of interrupts
* Add one to the count of an interrupt in the global state
* Clear the count of an interrupt in the global state
* Report whether an interrupt is enabled or not

For example, you should be able to do something:

|  |
| --- |
| intcon\_init( ); intcon\_enable(IRQ1); intcon\_enable(IRQ4);  …. intcon\_clearirq ( );  … kprintf(“There were %d interrupts\n”, intcon\_getcount(IRQ1); |

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. Show me your best, most well organized, well-documented code for the interrupt section
2. I want to see that you’ve taken the idea of interrupt handling and *made it easier* for other programmers, remember, one of the primary tasks of any OS is to create functionality from the base machine that wasn’t there originally.

Points: \_\_\_\_\_\_\_\_\_ / 50

1. https://pdos.csail.mit.edu/6.828/2014/readings/hardware/8259A.pdf [↑](#footnote-ref-1)