CS310

Operating Systems

Prof. Shitala Prasad

Project - MARN OS:

Mohammad Senan Ali -2104218

Aditya Rajesh Bawangade - 2103111

Rishabh Jain - 2104226

Nishant Kumar Singh - 2104221

Contents

Chapter 1

About MARN OS

We started this project from a codebase for a Learning Operating System which implements minimal memory management, text output, Global Descriptor Table (GDT), Programmable Interrupt Controller (PIC), Hardware abstraction level (HAL) among others.

1.1 Building an OS using Assembly and C for Intel x86 Machines

Figure 1.1: Intel Core 2 Duo, an example of an x86-compatible, 64-bit multicore processor

As an addition to this Operating System available on the internet (Github Link) we have implemented interrupts and interrupt handling using assembly and C programs. When an interrupt was triggered, the CPU saved its then-current state, including registers, the program counter, and flags, on the stack. The CPU looked up the ISR associated with the interrupt type in the IDT. Control was transferred to the ISR, which executed specific code to handle the interrupt or exception. After handling the interrupt, the ISR could have altered the saved state before issuing an IRET instruction, which restored the CPU's state and allowed it to continue executing the interrupted program.

Interrupt handlers were created to manage different types of interrupts, encompassing exceptions, hardware interrupts, and software interrupts. These handlers were implemented as functions to address specific interrupt scenarios, such as saving CPU states, executing relevant code, and returning control to the interrupted program. Assembly code was written to create interrupt handlers. These handlers functioned as special routines designed to meet specific requirements. They were responsible for saving the CPU state when an interrupt occurred, including registers, flags, and other relevant information. After handling the interrupt, the handlers used the IRET instruction to return control to the interrupted program or system.

Chapter 2

Interrupts and MARN O.S.

Question 1: Important

What is an Interrupt???

2.1 Interrupts

Interrupts are signals that interrupt the normal execution of a CPU to handle specific events, like hardware input or software requests. They help manage real-time tasks and ensure efficient multitasking. Interrupts serve as a way for external hardware or software to gain the attention of the CPU. When an interrupt occurs, it temporarily diverts the CPU's attention from its current task to handle the interrupting event. The CPU responds to interrupts by stopping its current operation, saving the current state (including the program counter and registers), and transferring control to an Interrupt Service Routine (ISR) or handler specific to the interrupt type. The ISR is a piece of code that performs tasks related to handling the interrupt. It can be a system-provided routine (for exceptions) or a custom one (for hardware or software interrupts). Interrupt controllers aggregate hardware interrupts and deliver them to the CPU. Interrupts are crucial for managing input, hardware devices, timers, and error conditions.

2.1.1 Software Interrupt

A software interrupt, often referred to as a *trap* or a *system call*, is a mechanism for a program to request services from the operating system. It allows user-level applications to transition to the kernel mode to perform privileged operations like I/O, memory management, or other system-related tasks. Software interrupts are initiated by specific instructions in the code and are essential for interfacing applications with the underlying operating system. Software interrupts, are a way for a program to communicate with the operating system or request specific services.

2.1.2 Hardware Interrupt

A hardware interrupt is an interrupt generated by external hardware devices to signal the CPU that they need immediate attention. These interrupts can be triggered by various hardware events, such as user input (keyboard or mouse events), timer or clock signals, disk I/O completion, or communication from peripheral devices. Hardware interrupts are crucial for real-time responsiveness and event-driven processing in a computer system. When a hardware interrupt occurs, the CPU temporarily suspends its current tasks and jumps to a specific interrupt handler to deal with the external event.

2.1.3 User-defined Interrupts

A user-defined interrupt is a type of interrupt in computing that is created and defined by the user or programmer to handle specific events or conditions within a program. Unlike hardware interrupts, which are generated by external devices or the system, user-defined interrupts are deliberately inserted into the code to pause the normal program flow and execute a designated routine or function in response to a particular situation. This allows users to have greater control and flexibility in managing and responding to events or conditions within their software applications.

2.2 Enhancing and extending capabilities for interrupt creation and interrupt handling

The codebase included an exception table as well as an interrupt handler array with void functions that would handle the interrupt, this was externed from an assembly programming running our OS. To enable the user we have added the feature and ability to define user defined interrupts, which have been used as mentioned later in this report. These interrupts can possibly enable the OS to give some extra functionality to the users. We have also worked on ways to create these interrupts by calling a crashing function in the main file and then handling it. The OS is able to handle some basic interrupts such as Division by Zero, Overflow, Timer, Keyboard and MARN interrupt (user defined interrupt), the details for which are mentioned in the upcoming chapter.

Chapter 3

Interrupt Descriptor Table

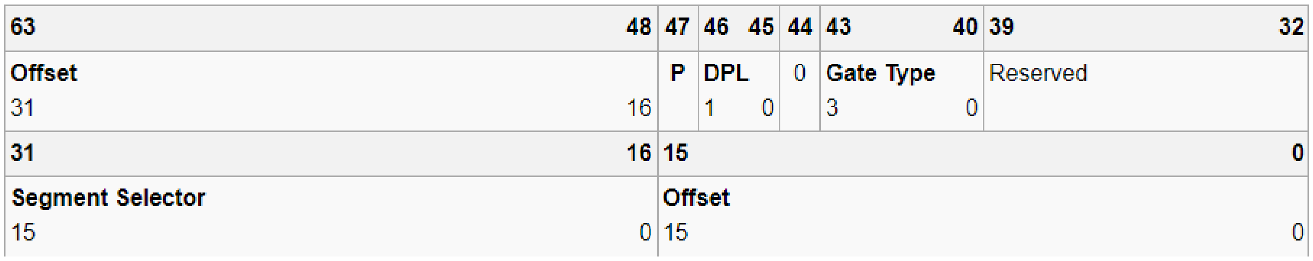
Definition 3.1: Interrupt

The Interrupt Descriptor Table (IDT) is a fundamental data structure in the x86 architecture. It functions as a mapping table for handling interrupts and exceptions, which can be triggered by various events in a computer system.

The IDT consists of a collection of entries, each of which corresponds to a specific interrupt or exception. These entries contain essential information, including the memory address of the code that manages the particular interrupt or exception, a code segment selector, and various flags indicating the privilege level and the type of the entry. When an interrupt or exception occurs, the CPU uses its unique number to find the corresponding entry in the IDT. This entry points to the location of the code, known as the Interrupt Service Routine (ISR), responsible for handling that specific event. The CPU then transfers control to the appropriate ISR. Additionally, the IDT entries provide information about the required privilege level to execute the associated ISR. This helps ensure that only authorized code can run certain interrupt service routines, enhancing system security and control. In summary, the IDT is a critical component of the x86 architecture, guiding the CPU in responding to interrupts and exceptions effectively, enabling the system to manage hardware and software events efficiently.

3.1 Implementation in x86

In 32 and 64 bit architectures the location of the IDT is stored in the register IDTR (Interrupt Descriptor Table Register).



63 **48**	**47**	**46** **45**	**44**	**43** **40**	**39** **32**
Offset 31 16	**P**	**DPL** 1 0	0	**Gate Type** 3 0	Reserved

31 **16**	**15** **0**
Segment Selector 15 0	**Offset** 15 0

Figure 3.1: IDT Entry

C Code For IDT Entry

```
typedef struct
{
    uint16_t BaseLow;
    uint16_t SegmentSelector;
    uint8_t Reserved;
    uint8_t Flags;
    uint16_t BaseHigh;
} __attribute__((packed)) IDTEntry;
```

- The `BaseLow` and `BaseHigh` stores the location of the handler function.
- `SegmentSelector` stores the location of the segment in the Global Descriptor Table (which will be our code segment).
- `Reserved` in this is always zero in this architecture.
- `flag` is used to specify more details about this entry.

C Code For IDT Table

```
typedef struct
{
    uint16_t Limit;
    IDTEntry* Ptr;
} __attribute__((packed)) IDTDescriptor;
```

- `Limit` is for declaring the size of the descriptor table.
- `Ptr` is a pointer to the table location.

Macros for Flags

```
typedef enum
{
    IDT_FLAG_GATE_TASK              = 0x5,
    IDT_FLAG_GATE_16BIT_INT         = 0x6,
    IDT_FLAG_GATE_16BIT_TRAP        = 0x7,
    IDT_FLAG_GATE_32BIT_INT         = 0xE,
    IDT_FLAG_GATE_32BIT_TRAP        = 0xF,

    IDT_FLAG_RING0                  = (0 << 5),
    IDT_FLAG_RING1                  = (1 << 5),
    IDT_FLAG_RING2                  = (2 << 5),
    IDT_FLAG_RING3                  = (3 << 5),
    IDT_FLAG_PRESENT                = 0x80,

} IDT_FLAGS;
```

`IDT_FLAG_GATE_TASK`: This flag is used to define a task gate. A task gate allows for switching between different tasks or threads when an interrupt occurs. It is primarily used for multitasking and is less common in modern operating systems.

`IDT_FLAG_GATE_16BIT_INT`: This flag is used to define a 16-bit interrupt gate. When an interrupt with this gate is triggered, the processor switches to privilege level 0 and executes a 16-bit interrupt handler.

`IDT_FLAG_GATE_16BIT_TRAP`: This flag is used to define a 16-bit trap gate. Similar to the 16-bit interrupt gate, but it doesn't disable interrupts during the handler's execution.

`IDT_FLAG_GATE_32BIT_INT`: This flag is used to define a 32-bit interrupt gate. When an interrupt with this gate is triggered, the processor switches to privilege level 0 and executes a 32-bit interrupt handler.

`IDT_FLAG_GATE_32BIT_TRAP`: This flag is used to define a 32-bit trap gate. Similar to the 32-bit interrupt gate, but it doesn't disable interrupts during the handler's execution.

`IDT_FLAG_RING0, IDT_FLAG_RING1, IDT_FLAG_RING2, IDT_FLAG_RING3`: These flags are used to specify the privilege level (ring level) at which the interrupt or exception handler should run. Ring 0 is the most privileged level, and Ring 3 is the least privileged. The ¡¡ 5 is used to shift the privilege level bits to the correct position within the flags.

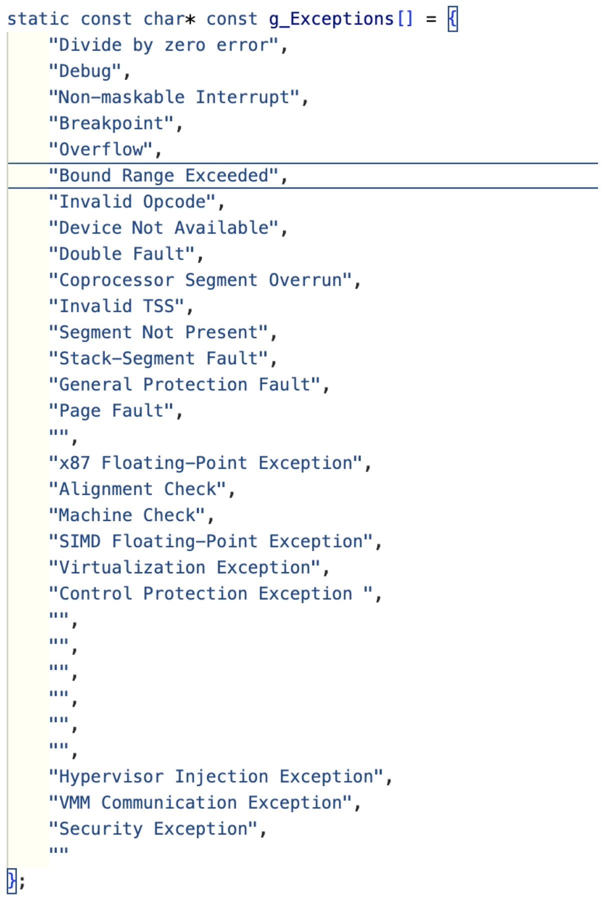
`IDT_FLAG_PRESENT`: This flag indicates whether the entry is present or not. If set (1), it means the entry is valid and present in the IDT; if not set (0), it's not used

C Code For Initialising IDT Entry

```
void i686_IDT_SetGate(int interrupt, void* base, uint16_t segmentDescriptor, uint8_t f
{
    g_IDT[interrupt].BaseLow = ((uint32_t)base) & 0xFFFF;
    g_IDT[interrupt].SegmentSelector = segmentDescriptor;
    g_IDT[interrupt].Reserved = 0;
    g_IDT[interrupt].Flags = flags;
    g_IDT[interrupt].BaseHigh = ((uint32_t)base >> 16) & 0xFFFF;
}
```

Chapter 4

Generating and Handling Exceptions using Interrupts



```
static const char* const g_Exceptions[] = {
    "Divide by zero error",
    "Debug",
    "Non-maskable Interrupt",
    "Breakpoint",
    "Overflow",
    "Bound Range Exceeded",
    "Invalid Opcode",
    "Device Not Available",
    "Double Fault",
    "Coprocessor Segment Overrun",
    "Invalid TSS",
    "Segment Not Present",
    "Stack-Segment Fault",
    "General Protection Fault",
    "Page Fault",
    "",
    "x87 Floating-Point Exception",
    "Alignment Check",
    "Machine Check",
    "SIMD Floating-Point Exception",
    "Virtualization Exception",
    "Control Protection Exception ",
    "",
    "",
    "",
    "",
    "",
    "",
    "Hypervisor Injection Exception",
    "VMM Communication Exception",
    "Security Exception",
    ""
};
```

Figure 4.1: Exceptions stored in IDT

4.1 Division by Zero

We have created a `subzero` subroutine in x86 ASM that we call to generate our division by zero error.

```
    global subzero
subzero:
    xor edx, edx      ; Clear the EDX register (for the remainder)
    mov eax, 9        ; Set EAX to the dividend (9)
    mov ebx, 0        ; Set EBX to the divisor (4)
    div ebx           ; Divide EAX by EBX, quotient in EAX, remainder in EDX

    mov ebx, 1
    div ebx

    ret
```

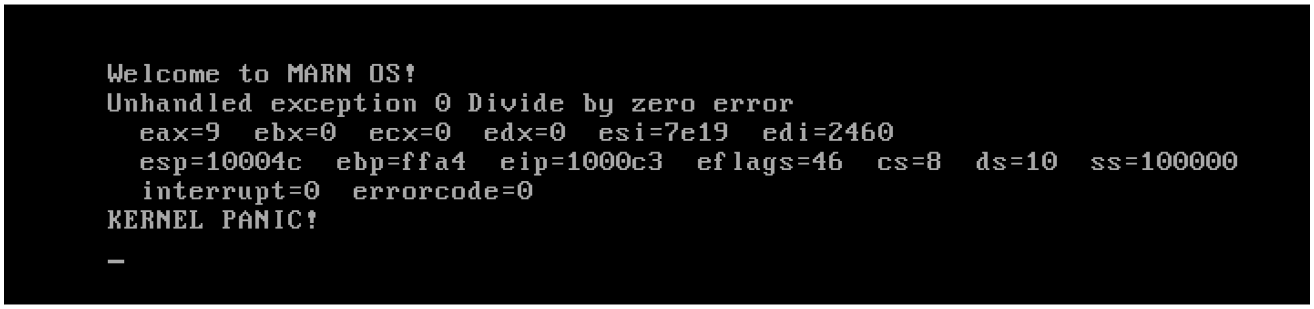
Unexpectedly, this generates 2 division by zero errors as opposed to simply 1 which would have been generated by

```
    mov ebx, 0        ; Set EBX to the divisor (4)
    div ebx           ; Divide EAX by EBX, quotient in EAX, remainder in EDX
```

The second division by zero error arises because we have not reset the value of `edx` register and the div operator divides `edx:eax` by `ebx` so in the segment

```
    mov ebx, 1
    div ebx
```

When we divide the value by 1 it is supposed to return the output `edx:eax` in `eax`, however it is a 64-bit value while `eax` is a 32-bit register which causes a Division by zero error, hence while handling the division by zero error we have to take this into consideration.

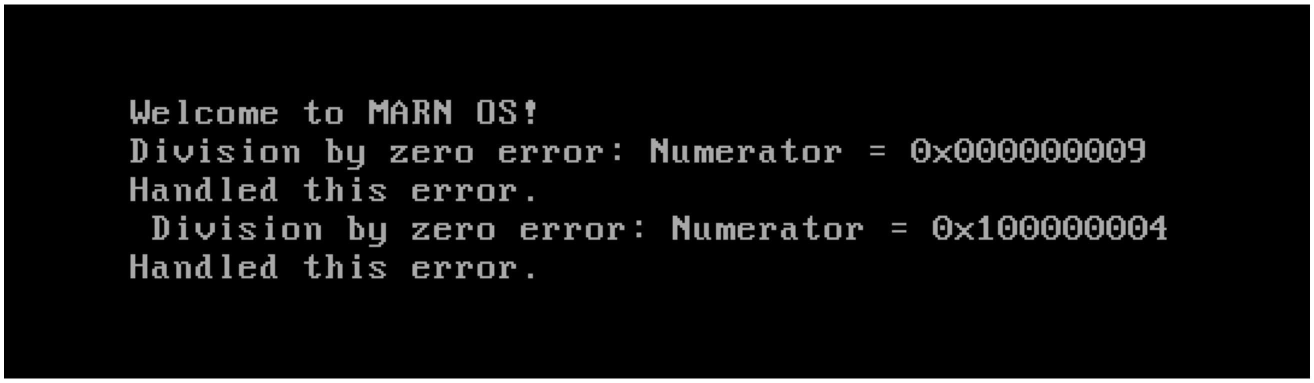


```
Welcome to MARN OS!
Unhandled exception 0 Divide by zero error
  eax=9  ebx=0  ecx=0  edx=0  esi=7e19  edi=2460
  esp=10004c  ebp=ffa4  eip=1000c3  eflags=46  cs=8  ds=10  ss=100000
  interrupt=0  errorcode=0
KERNEL PANIC!
_
```

Figure 4.2: Unhandled division exception

4.2 Handling Division by Zero Interrupt

To handle this division by zero interrupt, we have implemented an ISR Handler function divisionZero, this deals with both the cases for division by zero error by setting `edx` to 0 and `ebx` to 1. The output after handling is shown below:



```
Welcome to MARN OS!
Division by zero error: Numerator = 0x000000009
Handled this error.
 Division by zero error: Numerator = 0x100000004
Handled this error.
```

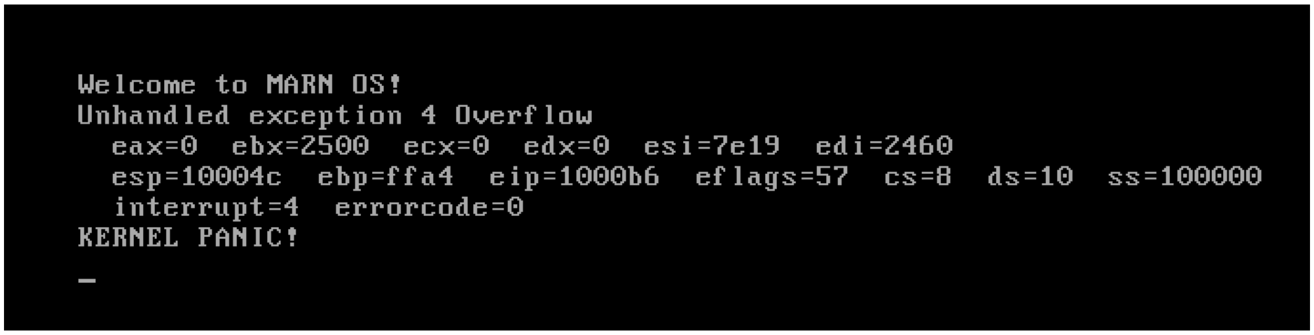
Figure 4.3: Handled division exception

4.3 Overflow Interrupt

We have created a `haddpar` subrouting which causes an overflow

```
global haddPaar
haddPaar:
    xor edx, edx
    mov eax, 0xFFFFFFFF; Load the value into the EAX register
    add eax, 1
    int 4
    ret
```

This is because we are adding `1` to `0xFFFFFFFF` resulting in an overflow.

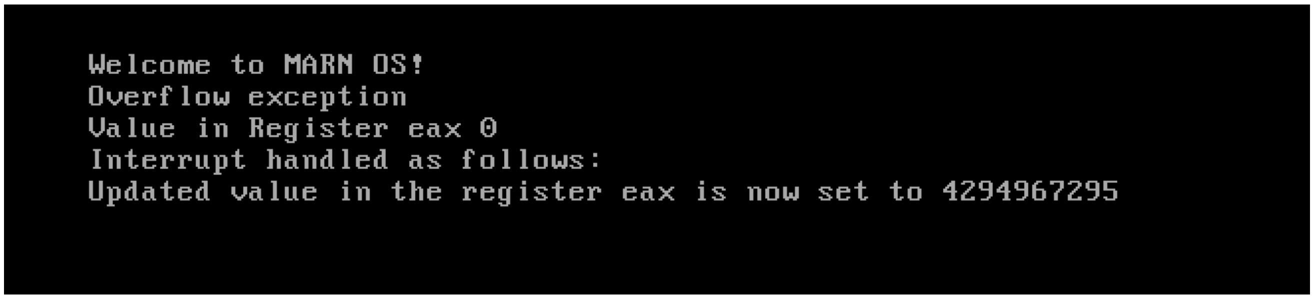


```
Welcome to MARN OS!
Unhandled exception 4 Overflow
  eax=0  ebx=2500  ecx=0  edx=0  esi=7e19  edi=2460
  esp=10004c  ebp=ffa4  eip=1000b6  eflags=57  cs=8  ds=10  ss=100000
  interrupt=4  errorcode=0
KERNEL PANIC!
_
```

Figure 4.4: Unhandled overflow exception

4.4 Handling Overflow Interrupt

To handle the overflow interrupt, we have implemented an ISR Handler function Overflow, this deals with the issue of overflowing by by resetting `eax` back to `0xFFFFFFFF`. Output is shown below

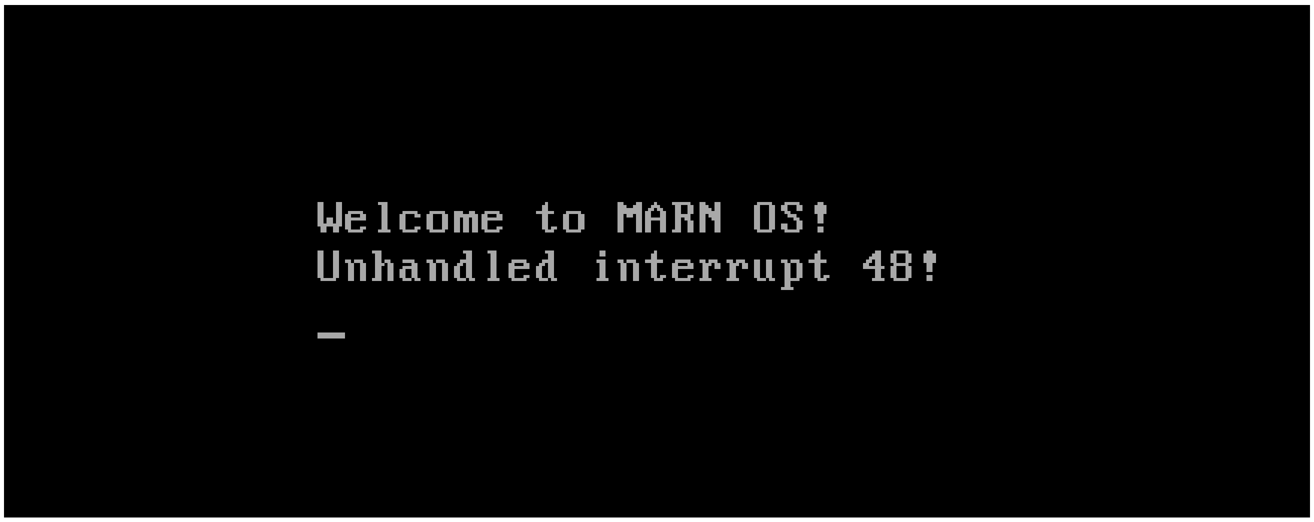


```
Welcome to MARN OS!
Overflow exception
Value in Register eax 0
Interrupt handled as follows:
Updated value in the register eax is now set to 4294967295
```

Figure 4.5: Unhandled overflow exception

4.5 User defined Interrupt

We can define interrupts for users to call, one such interrupt defined by us with IDT address 48, can be called by using inline assembly `__asm($0x30$)`. Output before handling is given below

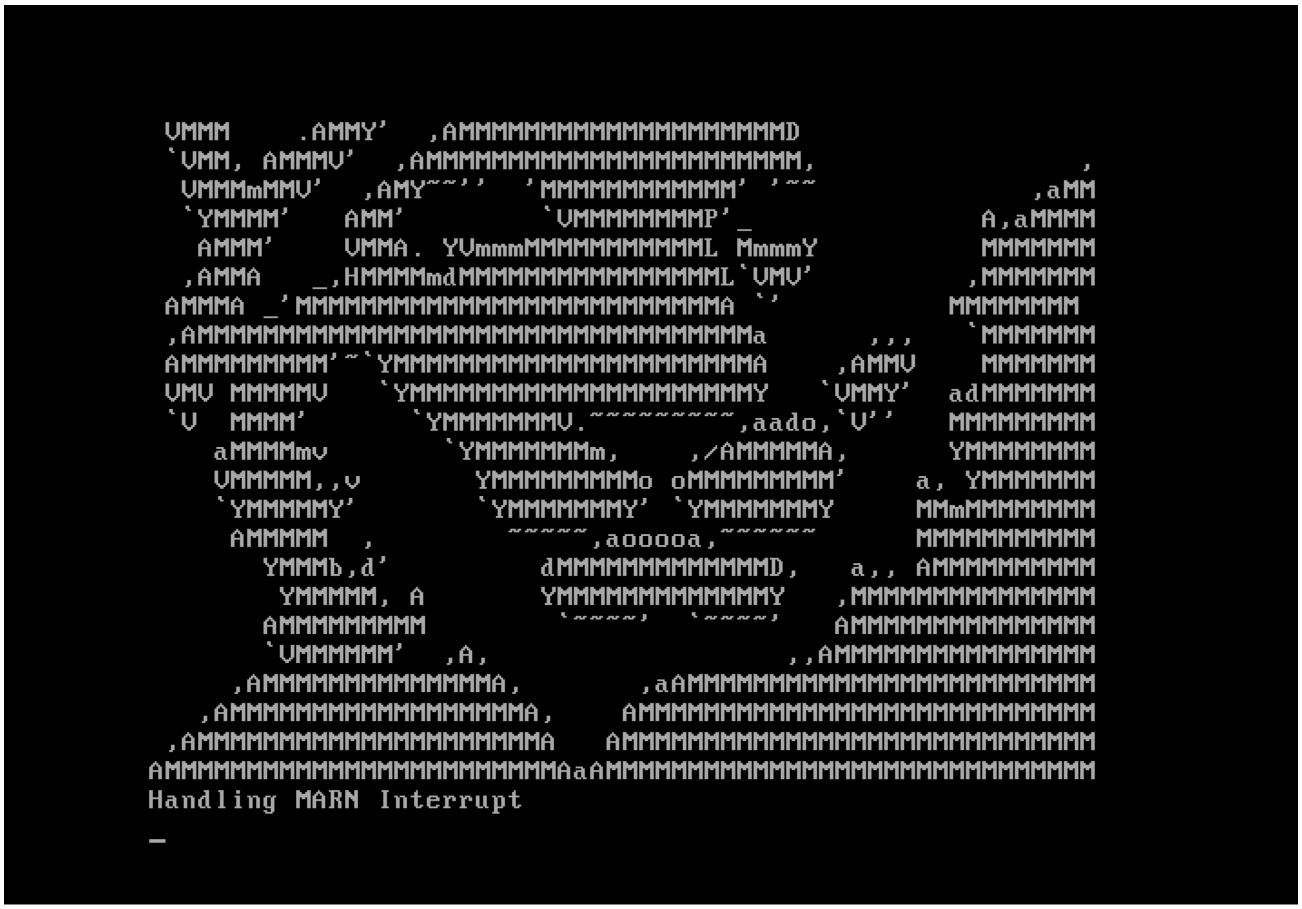


```
Welcome to MARN OS!
Unhandled interrupt 48!
_
```

Figure 4.6: Unhandled user interruption

4.6 Handling User defined interrupts

We can define the Handler as per our requirement, for this OS we have just implemented it as a function which prints a Lion on the screen. Output is shown below:



```
 VMMM    .AMMY' ,AMMMMMMMMMMMMMMMMMMMD
 `VMM, AMMMV' ,AMMMMMMMMMMMMMMMMMMMMMM,               ,
  VMMMmMMV'  ,AMY~~'' 'MMMMMMMMMMMM' '~~           ,aMM
  `YMMMM'  AMM'        `VMMMMMMMMP'_            A,aMMMM
   AMMM'   VMMA. YVmmmMMMMMMMMMML MmmmY         MMMMMMM
  ,AMMA   _,HMMMMmdMMMMMMMMMMMMMML`VMV'        ,MMMMMMM
 AMMMA _'MMMMMMMMMMMMMMMMMMMMMMMMA `'         MMMMMMMM
 ,AMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMa      ,,,  `MMMMMMM
 AMMMMMMMMM'~`YMMMMMMMMMMMMMMMMMMMMA    ,AMMV   MMMMMMM
 VMV MMMMMV   `YMMMMMMMMMMMMMMMMMMMY   `VMMY' adMMMMMMM
 `V  MMMM'     `YMMMMMMMV.~~~~~~~,aado,`V''   MMMMMMMMM
    aMMMMmv      `YMMMMMMMm,   ,/AMMMMMA,     YMMMMMMMM
    VMMMMM,,v     YMMMMMMMMMo oMMMMMMMMM'   a, YMMMMMMM
    `YMMMMMY'      `YMMMMMMMY' `YMMMMMMMY   MMmMMMMMMMM
     AMMMMM  ,       ~~~~~,aooooa,~~~~~~    MMMMMMMMMMM
       YMMMb,d'        dMMMMMMMMMMMMD,  a,, AMMMMMMMMMM
        YMMMMM, A      YMMMMMMMMMMMMY   ,MMMMMMMMMMMMMM
       AMMMMMMMMM      `~~~~~' `~~~~~'  AMMMMMMMMMMMMMM
       `VMMMMMM' ,A,                 ,,AMMMMMMMMMMMMMMM
     ,AMMMMMMMMMMMMMA,      ,aAMMMMMMMMMMMMMMMMMMMMMMMM
   ,AMMMMMMMMMMMMMMMMMA,   AMMMMMMMMMMMMMMMMMMMMMMMMMMM
 ,AMMMMMMMMMMMMMMMMMMMMA  AMMMMMMMMMMMMMMMMMMMMMMMMMMMM
AMMMMMMMMMMMMMMMMMMMMMMAaAMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
Handling MARN Interrupt
_
```

Figure 4.7: Handled user interruption

Chapter 5

Programmable Interrupt Controller

5.1 Overview

The x86 architecture traditionally used an 8259 PIC (Intel 8259A Programmable Interrupt Controller) or a more advanced APIC (Advanced Programmable Interrupt Controller) for multiprocessor systems. The PIC is responsible for handling hardware interrupts generated by peripherals such as keyboards, mice, timers, and other devices. These interrupts are crucial for allowing the CPU to respond to external events and ensure that the operating system and software can interact with hardware devices effectively. This PIC is connected to different hardware devices

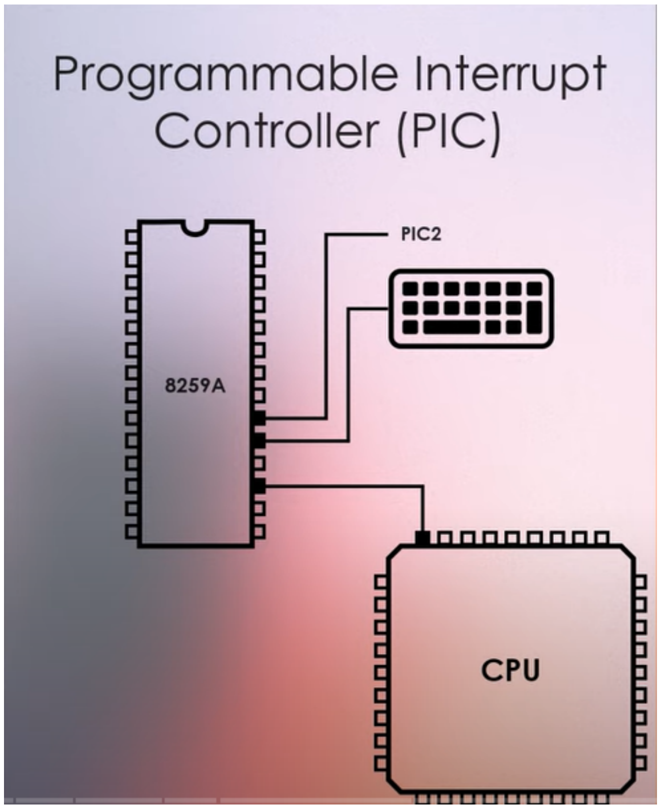


Figure 5.1: PIC connected to different hardware

through which interrupts can be detected

5.2 Key Functions and Features

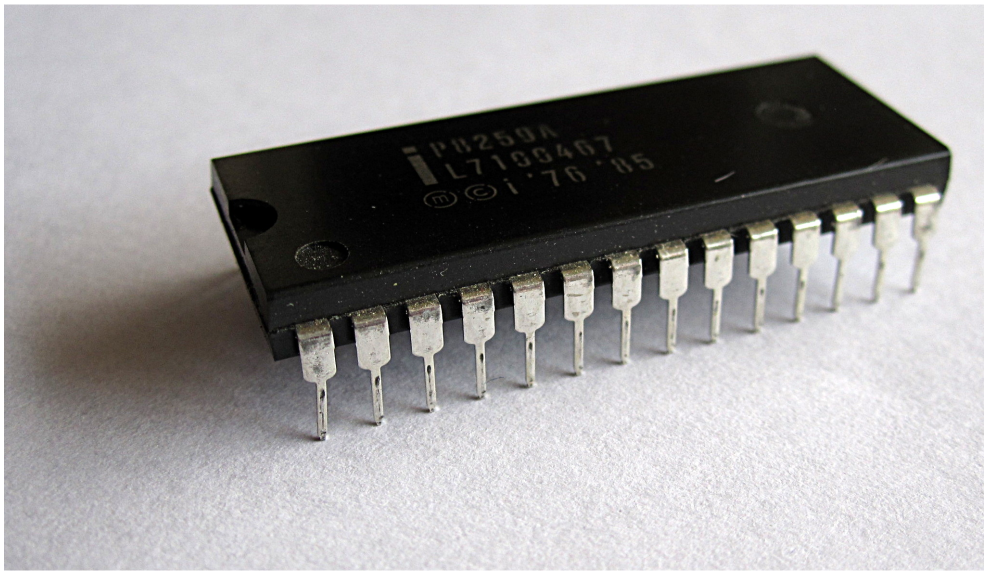


Figure 5.2: PIC

- **Interrupt Handling**: When a hardware device requires the CPU's attention (e.g., a key press on the keyboard), it generates an interrupt request (IRQ). The PIC plays a central role in prioritizing these requests and passing them to the CPU. It allows the CPU to respond to these interrupts in an orderly and managed way.
- **Interrupt Prioritization**: The PIC can manage multiple interrupt sources, each assigned a unique interrupt request line (IRQ). These sources can have different priority levels, ensuring that high-priority devices are serviced first.
- **Cascading**: In systems with multiple PICs, a master-slave configuration is often used. The master PIC manages the high-priority interrupts and communicates with the slave PIC. This allows for more IRQ lines and ensures that a wide range of devices can be accommodated.
- **Masking Interrupts**: The PIC provides the capability to mask or disable specific interrupt lines. This can be useful when certain hardware interrupts are not needed at a particular moment or to prevent interference with critical tasks.
- **Interrupt Acknowledgment**: The CPU communicates with the PIC to acknowledge and process interrupts. This acknowledgment process varies depending on whether the system uses an 8259 PIC or an APIC.
- **Initialization**: During system boot-up, the PIC needs to be initialized to configure its operation. This initialization process sets up interrupt vectors, priorities, and other settings.

C Code For PIC

```
#define PIC1_COMMAND_PORT           0x20
#define PIC1_DATA_PORT              0x21
#define PIC2_COMMAND_PORT           0xA0
#define PIC2_DATA_PORT              0xA1
```

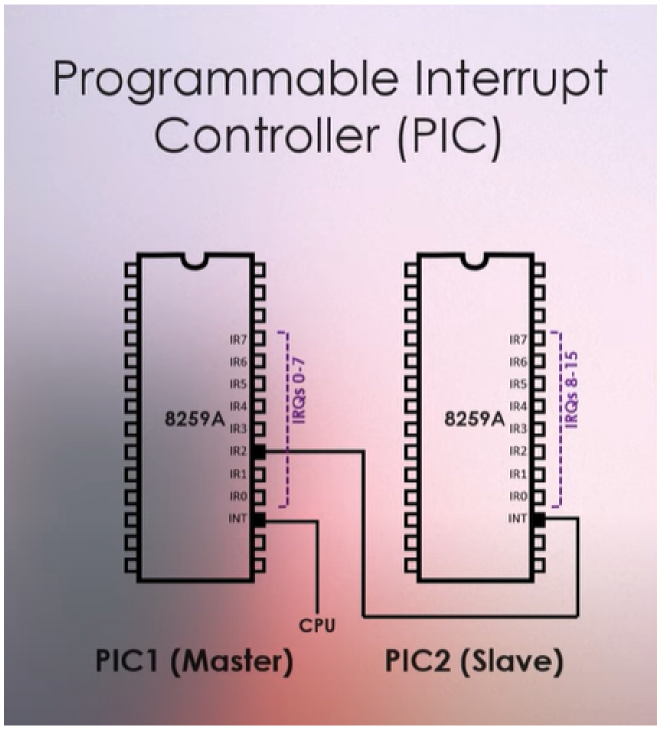


Figure 5.3: Master Slave Model

Macros For PIC

```
enum {
    PIC_ICW1_ICW4           = 0x01,
    PIC_ICW1_SINGLE         = 0x02,
    PIC_ICW1_INTERVAL4      = 0x04,
    PIC_ICW1_LEVEL          = 0x08,
    PIC_ICW1_INITIALIZE     = 0x10
} PIC_ICW1;
```

The 8259A accepts two types of command words generated by the CPU.

1. Initialization Command Words (ICWs): Before normal operation can begin, each 8259A in the system must be brought to a starting point.
2. Operation Command Words (OCWs): These are the command words which command the 8259A to operate in various interrupt modes.

- Fully nested mode
- Rotating priority mode
- Special mask mode
- Polled mode

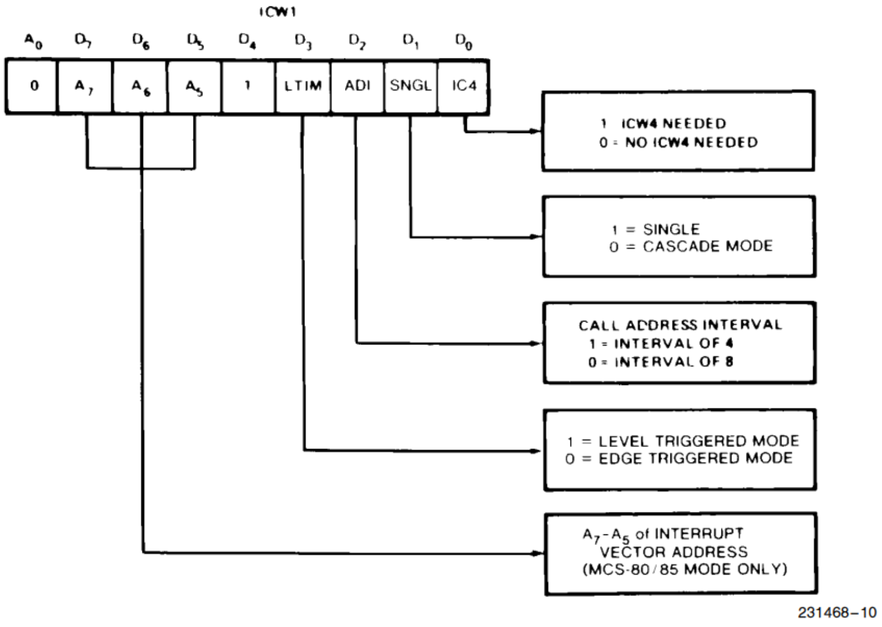


Figure 5.4: ICW1 from the Intel Manual

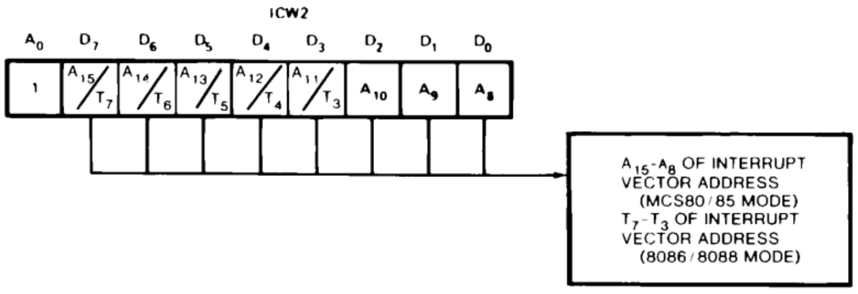

ICW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 A15/T7 A14/T6 A13/T5 A12/T4 A11/T3 A10 A9 A8
A15-A8 OF INTERRUPT VECTOR ADDRESS (MCS80/85 MODE)
T7-T3 OF INTERRUPT VECTOR ADDRESS (8086/8088 MODE)

Figure 5.5: ICW2 from the Intel Manual

$A_0 = 0$ signifies a that the input should be sent to a command port, whereas $A_0 = 1$ signifies the data is going to the data port.

```
enum {
    PIC_ICW1_ICW4           = 0x01,
    PIC_ICW1_SINGLE         = 0x00,
    PIC_ICW1_INTERVAL4      = 0x04,
    PIC_ICW1_LEVEL          = 0x08,
```

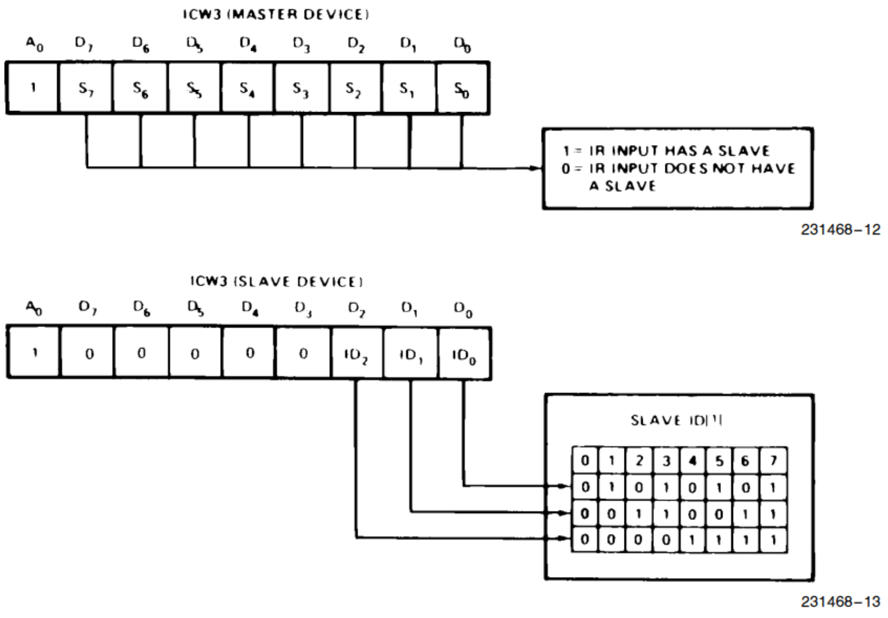


Figure 5.6: ICW3 from the Intel Manual

```
    PIC_ICW1_INITIALIZE     = 0x10
} PIC_ICW1;
```

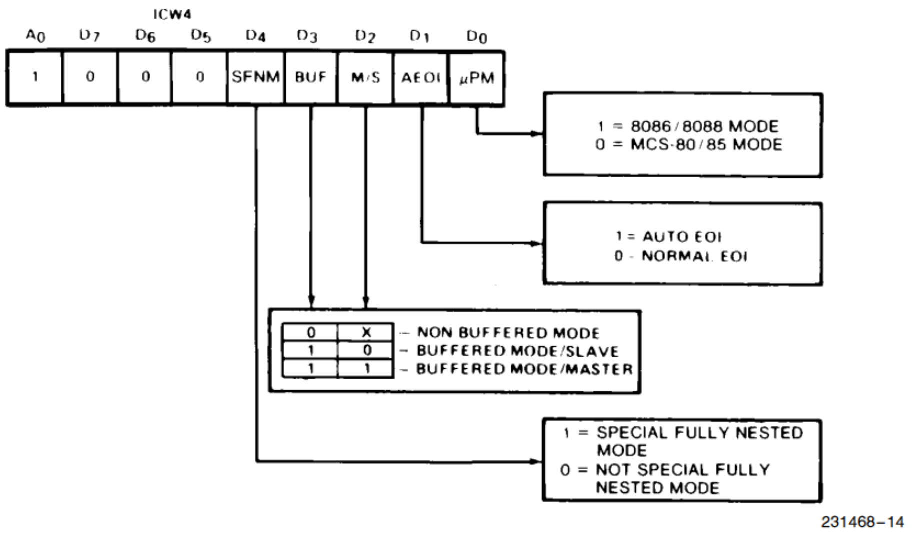

ICW4
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 0 0 0 SFNM BUF M/S AEOI μPM
1 = 8086/8088 MODE
0 = MCS-80/85 MODE
1 = AUTO EOI
0 - NORMAL EOI
0 X - NON BUFFERED MODE
1 0 - BUFFERED MODE/SLAVE
1 1 - BUFFERED MODE/MASTER
1 = SPECIAL FULLY NESTED MODE
0 = NOT SPECIAL FULLY NESTED MODE
231468-14

Figure 5.7: ICW4 from the Intel Manual

Let us configure the PIC

```
void i686_PIC_Configure(uint8_t offsetPic1, uint8_t offsetPic2)
{
    // initialization control word 1
    i686_outb(PIC1_COMMAND_PORT, PIC_ICW1_ICW4 | PIC_ICW1_INITIALIZE);
    i686_iowait();
    i686_outb(PIC2_COMMAND_PORT, PIC_ICW1_ICW4 | PIC_ICW1_INITIALIZE);
    i686_iowait();

    // initialization control word 2 - the offsets
    i686_outb(PIC1_DATA_PORT, offsetPic1);
    i686_iowait();
    i686_outb(PIC2_DATA_PORT, offsetPic2);
    i686_iowait();

    // initialization control word 3
    i686_outb(PIC1_DATA_PORT, 0x4);             // tell PIC1 that it has a
                                                // slave at IRQ2 (0000 0100)
    i686_iowait();
    i686_outb(PIC2_DATA_PORT, 0x2);             // tell PIC2 its cascade
                                                // identity (0000 0010)
    i686_iowait();

    // initialization control word 4
    i686_outb(PIC1_DATA_PORT, PIC_ICW4_8086);
    i686_iowait();
    i686_outb(PIC2_DATA_PORT, PIC_ICW4_8086);
    i686_iowait();
    // clear data registers
    i686_outb(PIC1_DATA_PORT, 0);
    i686_iowait();
    i686_outb(PIC2_DATA_PORT, 0);
    i686_iowait();
}
```

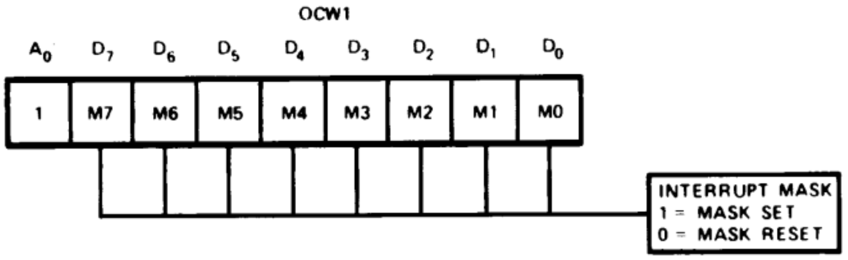


Figure 5.8: OCW1 from the Intel Manual

OCW1 is used to mask out the interrupts. The values of $M_0 - M_7$ is stored in a special register IMR - Interrupt Mask Register.

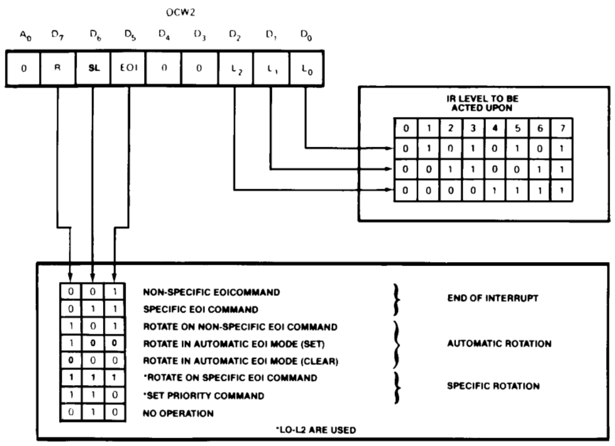

OCW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 R SL EOI 0 0 L2 L1 L0
IR LEVEL TO BE ACTED UPON
0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 1
0 0 1 NON-SPECIFIC EOICOMMAND
0 1 1 SPECIFIC EOI COMMAND
1 0 1 ROTATE ON NON-SPECIFIC EOI COMMAND
1 0 0 ROTATE IN AUTOMATIC EOI MODE (SET)
0 0 0 ROTATE IN AUTOMATIC EOI MODE (CLEAR)
1 1 1 *ROTATE ON SPECIFIC EOI COMMAND
1 1 0 *SET PRIORITY COMMAND
0 1 0 NO OPERATION
END OF INTERRUPT
AUTOMATIC ROTATION
SPECIFIC ROTATION
*LO-L2 ARE USED

Figure 5.9: OCW2 from the Intel Manual

Rotate in Automatic End of Interrupt (EOI): This helps in managing interrupt priorities.
Specific EOI: It allows you to send an EOI command to a specific interrupt level.
Non-specific EOI: This is used for acknowledging the highest priority interrupt. The RR

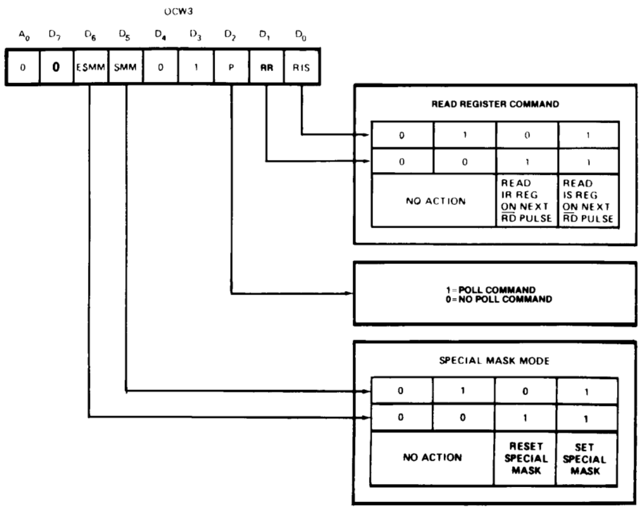


Figure 5.10: OCW3 from the Intel Manual

and RIS are used to read the internal registers of the PIC, it stores the data of the currently serviced interrupt, and and pending interrupts.

Polling is irrelevant for the MARN OS

Special Mask Mode is for by-passing priorities for certain interrupts.

```
void i686_PIC_SendEndOfInterrupt(int irq)
{
    if (irq >= 8)
        i686_outb(PIC2_COMMAND_PORT, PIC_CMD_END_OF_INTERRUPT);
    i686_outb(PIC1_COMMAND_PORT, PIC_CMD_END_OF_INTERRUPT);
}

void i686_PIC_Disable()
{
    i686_outb(PIC1_DATA_PORT, 0xFF);        // mask all
    i686_iowait();
    i686_outb(PIC2_DATA_PORT, 0xFF);        // mask all
    i686_iowait();
}

void i686_PIC_Mask(int irq)
{
    uint8_t port;

    if (irq < 8)
    {
        port = PIC1_DATA_PORT;
    }
    else
    {
        irq -= 8;
        port = PIC2_DATA_PORT;
    }

    uint8_t mask = i686_inb(PIC1_DATA_PORT);
    i686_outb(PIC1_DATA_PORT,  mask | (1 << irq));
}

void i686_PIC_Unmask(int irq)
{
    uint8_t port;

    if (irq < 8)
    {
        port = PIC1_DATA_PORT;
    }
    else
    {
        irq -= 8;
        port = PIC2_DATA_PORT;
    }

    uint8_t mask = i686_inb(PIC1_DATA_PORT);
    i686_outb(PIC1_DATA_PORT,  mask & ~(1 << irq));
}

uint16_t i686_PIC_ReadIrqRequestRegister()
{
    i686_outb(PIC1_COMMAND_PORT, PIC_CMD_READ_IRR);
    i686_outb(PIC2_COMMAND_PORT, PIC_CMD_READ_IRR);
    return ((uint16_t)i686_inb(PIC2_COMMAND_PORT)) |
    (((uint16_t)i686_inb(PIC2_COMMAND_PORT)) << 8);
}

uint16_t i686_PIC_ReadInServiceRegister()
{
    i686_outb(PIC1_COMMAND_PORT, PIC_CMD_READ_ISR);
    i686_outb(PIC2_COMMAND_PORT, PIC_CMD_READ_ISR);
    return ((uint16_t)i686_inb(PIC2_COMMAND_PORT)) |
    (((uint16_t)i686_inb(PIC2_COMMAND_PORT)) << 8);
}
```

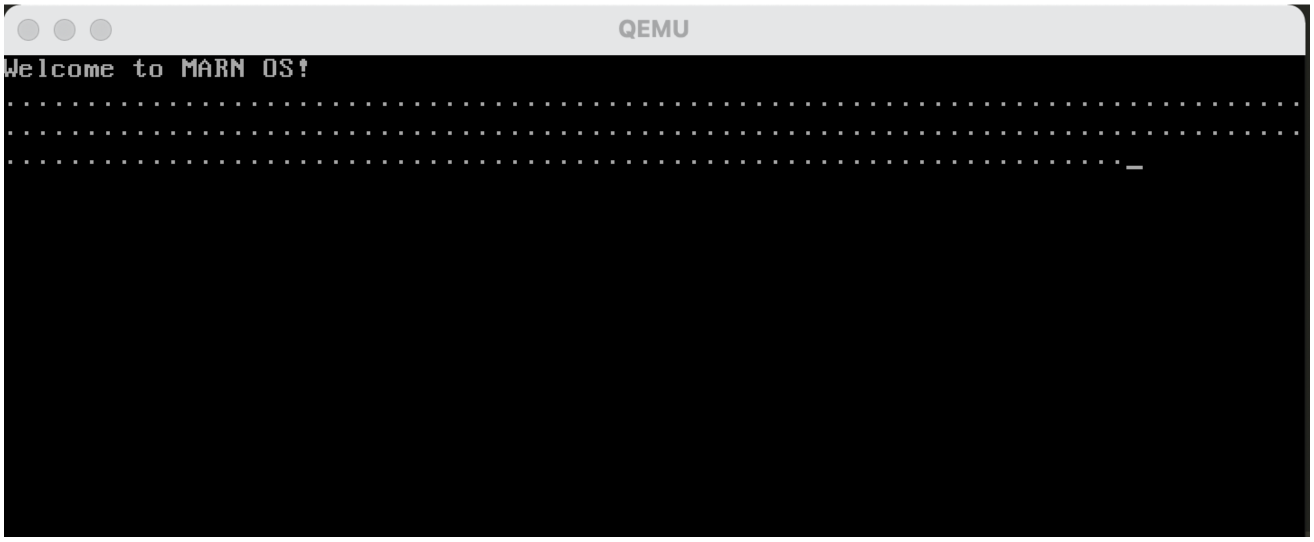
Chapter 6

Working with Hardware Interrupts

Hardware interrupts are detected by Interrupt requests (IRQ). These range from IRQ0 to IRQ15. Each IRQ corresponds to a hardware device which enables us to handle the Interrupt and parse the input. In out program we are dealing with two types of interrupts, namely Timer (IRQ0) and Keyboard interrupt (IRQ1). These interrupts are essential for parsing input in an Operating System

6.1 Timer (IRQ0)

Timer interrupts are crucial for multitasking and time-sensitive operations, such as scheduling tasks, updating system time, and maintaining accurate time measurements. They often have a fixed frequency, generating interrupts at regular intervals (e.g., every 1 millisecond) to maintain accurate timekeeping. Below is the output after handling timer interrupts:

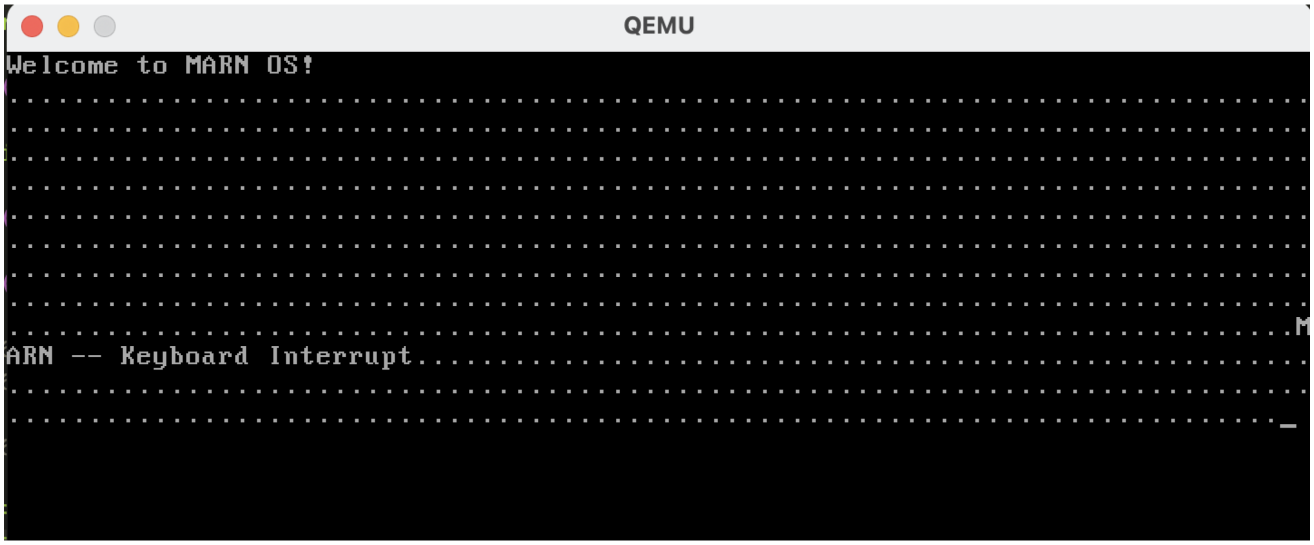


```
Welcome to MARN OS!
................................................................................
................................................................................
.....................................................................
```

Figure 6.1: Timer interruption

6.2 Keyboard (IRQ1)

Keyboard interrupts are generated when a user presses a key on the computer's keyboard or interacts with input devices like a mouse. These interrupts are essential for capturing user input and allowing the computer to respond to keystrokes or mouse movements in real-time. When a key is pressed or a mouse button is clicked, the corresponding hardware generates an interrupt signal to inform the CPU. Keyboard interrupts are handled by the operating system or specific device drivers, which translate the hardware-level signals into meaningful characters or commands. Below is the output after handling keyboard inputs:



```
Welcome to MARN OS!
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
...............................................................................M
ARN -- Keyboard Interrupt.......................................................
................................................................................
.............................................................................._
```

Figure 6.2: Keyboard interruption

Chapter 7

Conclusions

- The system implemented interrupt handling using assembly and C, including user-defined interrupts, exceptions, and hardware interrupts.
- The Interrupt Descriptor Table (IDT) played a crucial role in managing and routing interrupts and exceptions.
- Exception handling was demonstrated for scenarios like division by zero and overflow errors.
- The Programmable Interrupt Controller (PIC) was discussed, responsible for managing hardware interrupts and ensuring the CPU responds to external events.
- Timer (IRQ0) and keyboard (IRQ1) hardware interrupts were explained, essential for multitasking and user input capture.
- Overall, the MARN OS project showcased the importance of interrupt handling and hardware interaction in operating system development.

Chapter 8

Resources

- MIT Resources
- x86 Architecture resources
- x86 Architecture resources2
- x86 Architecture resources3
- CodeBase