MARN OS

M Mohammad Senan Ali

A Aditya Rajesh Bawangade

R Rishabh Jain

N Nishant Kumar Singh

ABOUT MARN OS

We initiated this project using an existing codebase for a Learning Operating System that encompasses essential features, such as basic memory management, text output, Global Descriptor Table (GDT) setup, Peripheral Interface Controller (PIC) configuration, and Hardware Abstraction Layer (HAL), among other functionalities.

Interrupt in OS

An interrupt is a signal to the processor generated by hardware or software indicating an event that needs immediate attention. It requires the operating system (OS) to stop and figure out what to do next. An interrupt temporarily stops or terminates a service or a current process.

Types of Interrupts:

Hardware Interrupts: Triggered by external hardware devices.
Software Interrupts: Generated by a program requiring OS intervention.

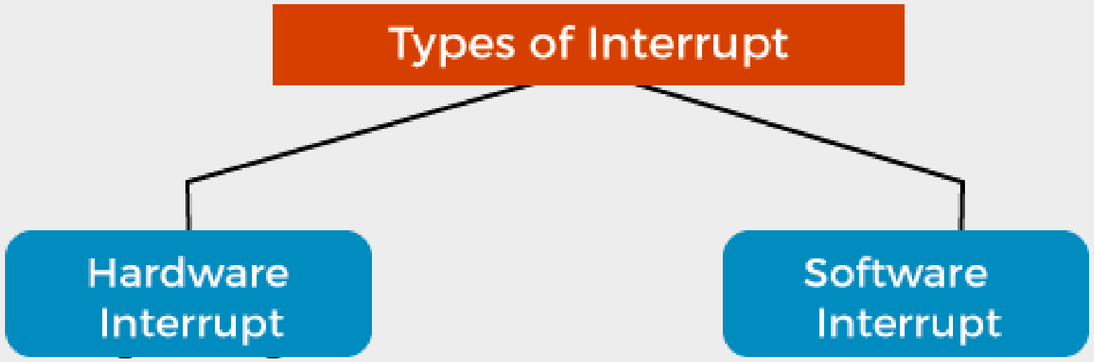
Purpose :

Interrupts are crucial for multitasking, allowing the CPU to respond to external events while executing other tasks

How Interrupts Work in OS

Interrupt Handling: When an interrupt occurs, the CPU stops its current tasks, saves its state, and executes a function called an Interrupt Service Routine (ISR).

Context Switching: This involves the OS saving the state of the current task and loading the state of the next task to be processed.



Real-World Examples of Interrupts

Divide By Zero exception:

- Definition: The Divide By Zero exception is a type of software interrupt or exception that occurs when a program attempts to divide a number by zero.
- CPU's Role: When the CPU detects a division operation with a zero divisor, it immediately generates an interrupt signal, halting the normal execution flow of the program.

Real-World Examples of Interrupts

User-Defined Interrupt:

Description: Triggered by user programs rather than hardware or system events.

Implementation: Typically implemented through system calls. The program issues a command, which is then interpreted by the OS as an interrupt, prompting a switch from user mode to kernel mode to execute the necessary operations.

Real-World Examples of Interrupts

Timer Interrupt:

Description: A Timer Interrupt is triggered by the system's hardware clock at regular intervals, preset by the operating system. Its primary function is to allow the operating system to perform regular tasks, such as updating system time, running scheduled tasks, or managing time-sharing among processes.

System Performance: The frequency of Timer Interrupts can impact overall system performance and responsiveness. A high frequency allows for more responsive task switching but can increase overhead due to frequent context switches.

Real-World Examples of Interrupts

Keyboard Interrupt:

Description: A Keyboard Interrupt is generated whenever a key is pressed or released on the keyboard. It signals the processor to stop its current activities and execute an Interrupt Service Routine (ISR) to handle the key press.

Input Handling: The ISR typically reads the scan code of the key pressed and either stores it in a buffer or processes it directly, depending on the OS design.

Intel x86 Architecture

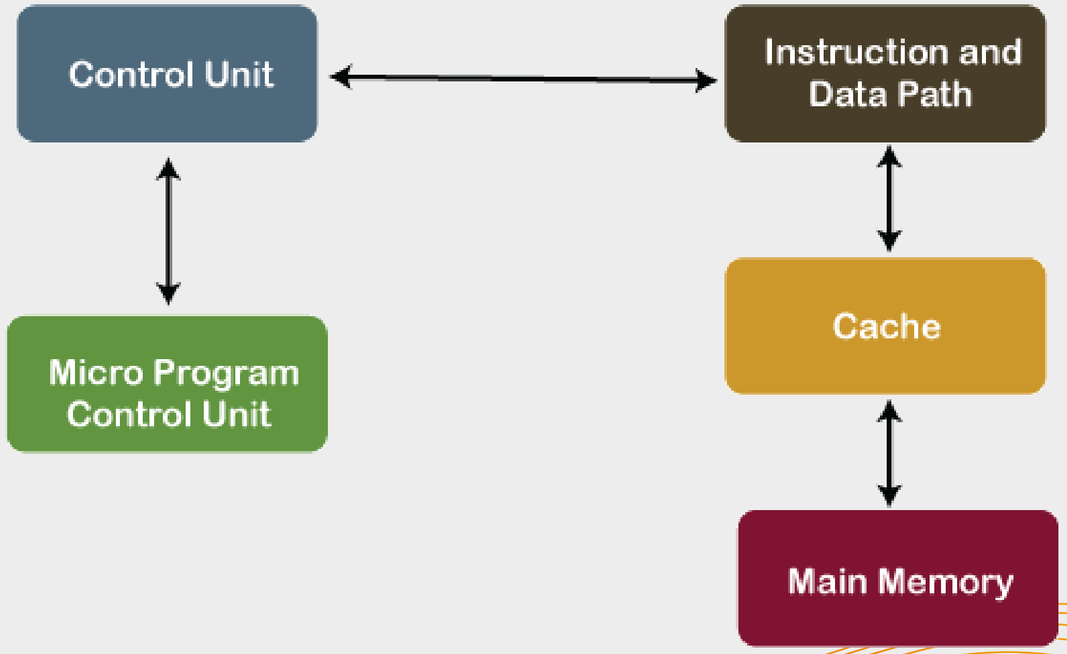
The Intel x86 architecture is a widely used and versatile computer processor architecture. It features a complex instruction set, supports both 32-bit and 64-bit modes, and provides compatibility with a wide range of software. It includes components like registers, memory addressing modes, and a variety of instructions for data manipulation and control flow.

The underlying codebase for this operating system is based on x86 Architecture.

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

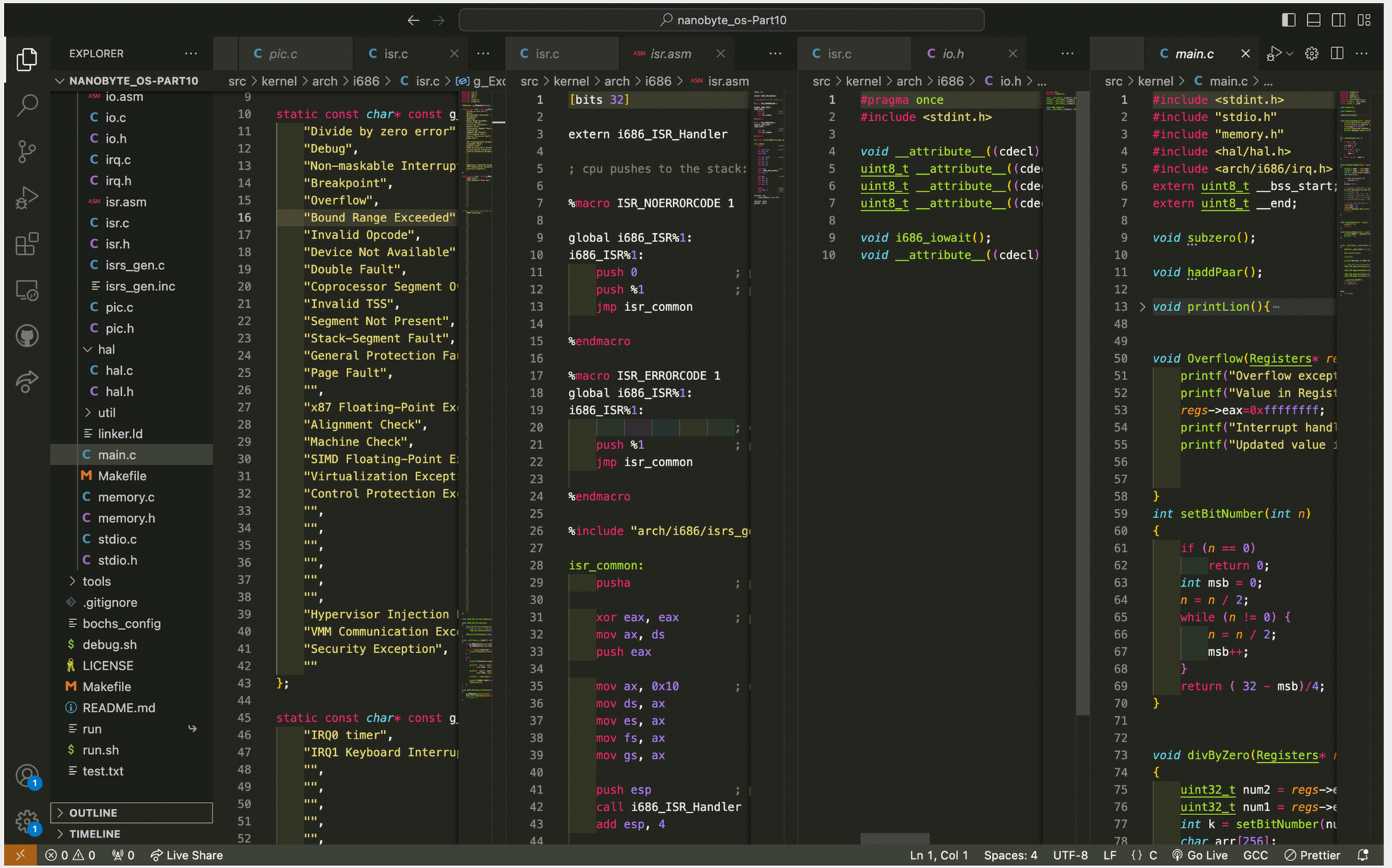
Intel x86 Architecture

1. CISC architecture
2. Registers: EAX, EBX, ECX, etc.
3. Memory Segmentation: Historical feature
4. Protected Mode: Introduced with 80286
5. 32-bit and 64-bit support
6. MMX: Multimedia SIMD instructions
7. SSE: Enhanced multimedia processing
8. Hyper-Threading: Simulates multiple processors
9. VT (Virtualization Technology): Hardware-level virtualization
10. Cache Levels: L1, L2, sometimes L3
11. Pipeline Architecture: Increases instruction throughput
12. Out-of-Order Execution: Improves efficiency
13. Power Management: Reduces energy consumption
14. FPU: Handles floating-point operations
15. Execution Units: Simultaneous instruction execution
16. AVX: Parallel data processing
17. NX Bit: Security feature
18. AES-NI: Accelerates encryption
19. Turbo Boost: Dynamic clock speed increase
20. Integrated Graphics: Some processors have it



CISC Architecture

Creating the MARN OS



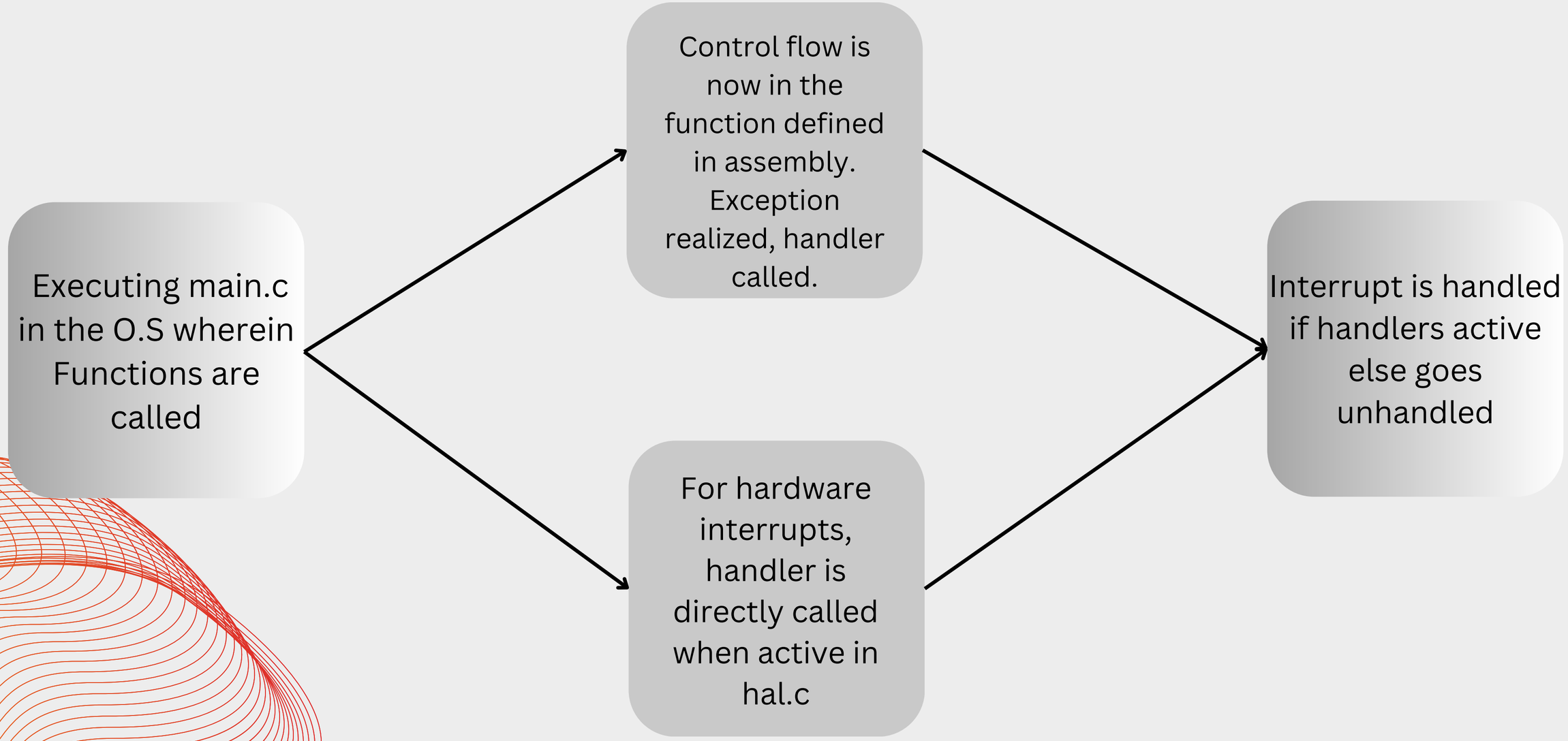
Interrupts and MARN OS

We enhanced the Operating System with interrupt handling, using assembly and C, allowing the CPU to save its state and execute specific code when interrupts occurred, then return to normal execution with the "IRET" instruction. Different handlers managed various interrupt types, ensuring proper handling and recovery.

```c
void divByZero(Registers* regs)
{
    uint32_t num2 = regs->eax;
    uint32_t num1 = regs->edx;
    int k = setBitNumber(num2);
    char arr[256];

    for(int i=0;i<k;i++){
        arr[i]='0';
    }
    arr[k]='\0';

    // uint32_t numerator = regs->ds;
    printf("Division by zero error: Numerator = 0x%x%s%x\n", num1,arr, num2);

    regs->ebx = 2;
    regs->edx = 0;
    printf("Handled this error.\n ");
    return ;
}
```

Understanding the Control Flow

The control flow of any interrupt is as follows:



Interrupt Descriptor Table (IDT)

The Interrupt Descriptor Table (IDT) is a crucial data structure within the x86 architecture, serving as a mapping table responsible for managing interrupts and exceptions. These interruptions can arise from a variety of events in a computer system.

Interrupt Descriptor Table (IDT)

IDT entries in 64Bit format

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

Defining IDT Entry in C

C code:

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

Defining IDT Table in C

C code:

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

Defining Macros for Flags in C

C code

```
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;
```

Initializing IDT Entry in C

C code

```
void i686_IDT_SetGate(int interrupt, void* base, uint16_t segmentDescriptor, uint8_t
{
    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;
}
```

Exceptions Stored in IDT

C code

```
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",
```

Creating and Handling Exceptions

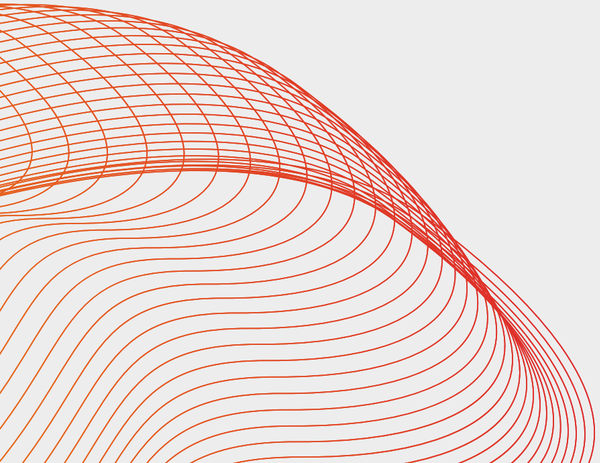
Assembly code for the subzero subroutine

```
    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
```

Unhandled Exceptions

State of the O.S. in a divide by zero Unhandled Exception

```
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!
_
```



Handling the Exception

State of the O.S. in a divide by zero Handled Exception

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

Overflow Interrupt

Causing the Overflow using Assembly Program

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

Overflow Interrupt

State of the O.S. in an Unhandled Overflow Interrupt

```
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!
_
```

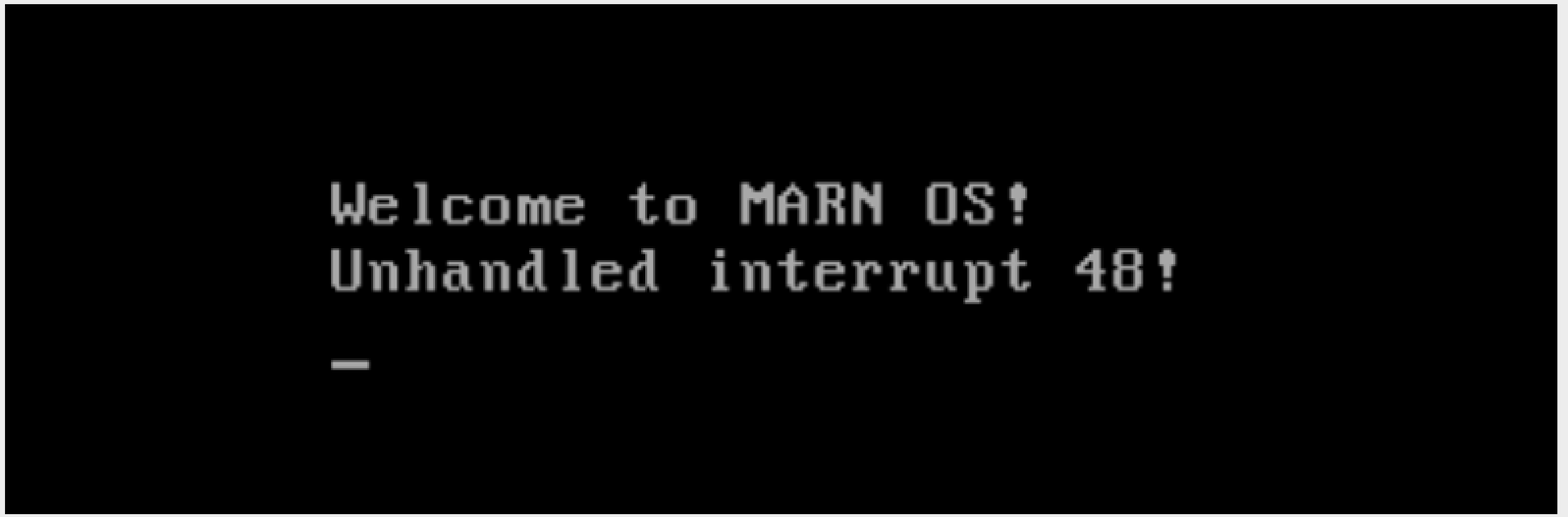
Overflow Interrupt

State of the O.S. In handled Overflow Interrupt

```
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
```

User-Defined Interrupt

State of the O.S. in an Unhandled User-Defined Interrupt

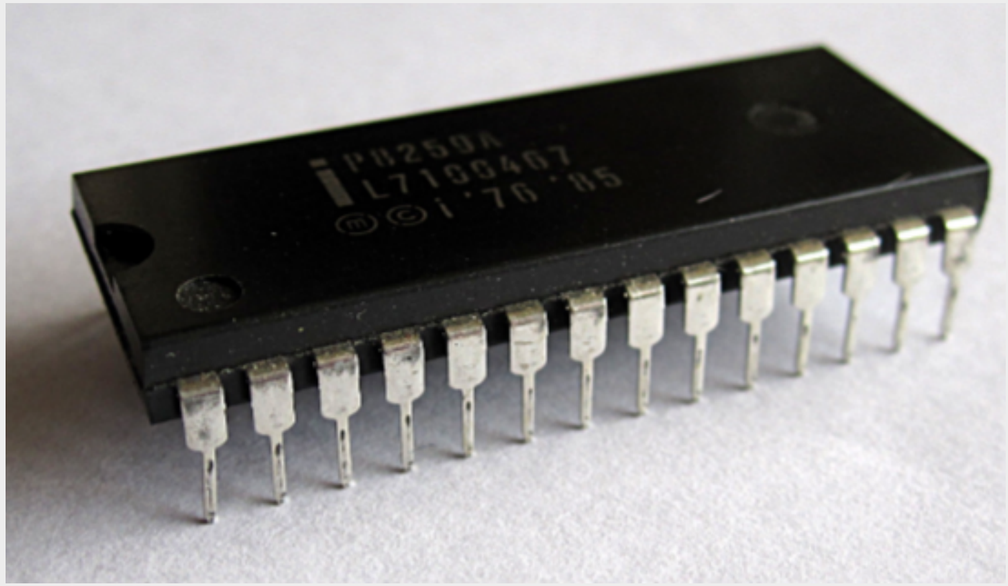


User-Defined Interrupt

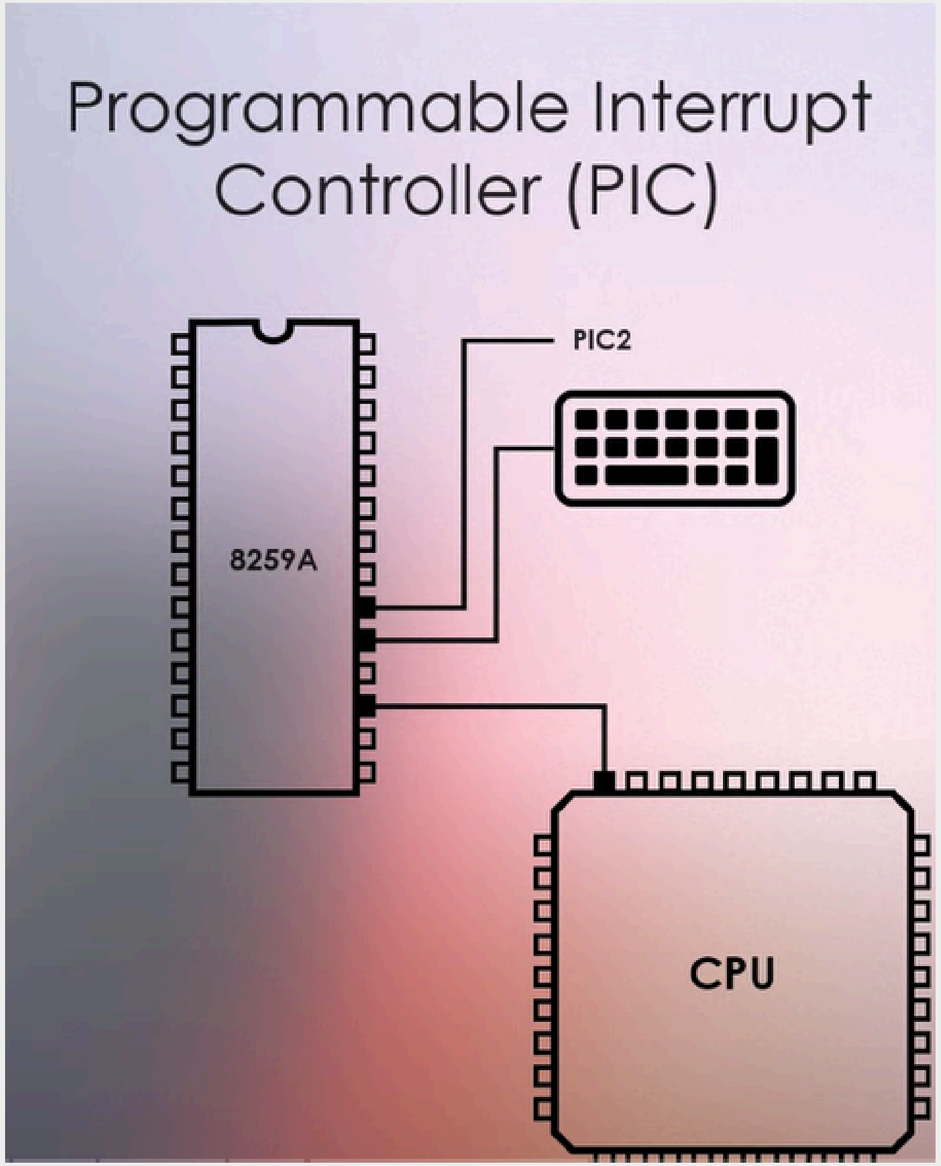
State of the O.S. in a handled User-Defined Interrupt

Programmable Interrupt Controller

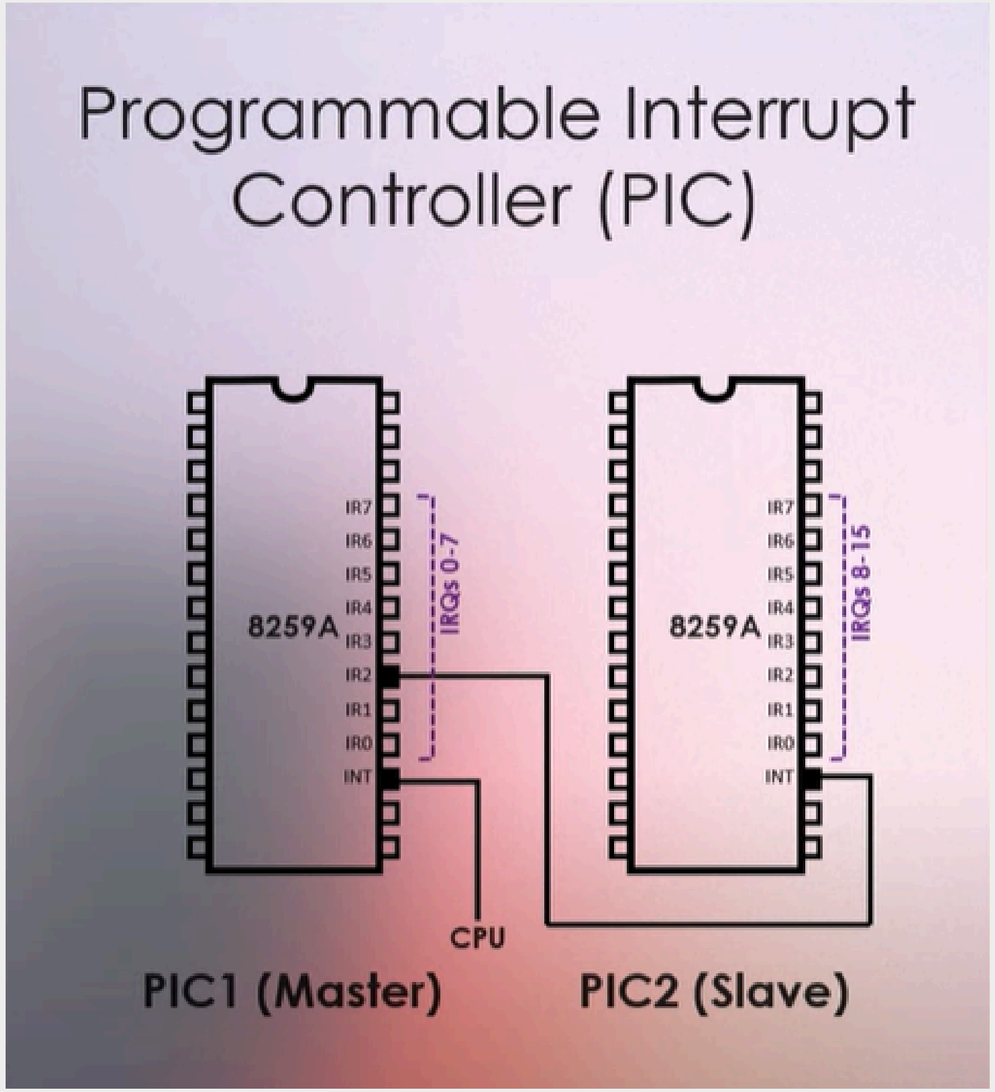
The x86 architecture traditionally used an 8259 PIC (Intel 8259A Programmable Interrupt Controller) or a more advanced APIC (Advanced Programmable Interrupt Controller) for multiprocessor systems.



Programmable Interrupt Controller



Programmable Interrupt Controller



Programmable Interrupt Controller

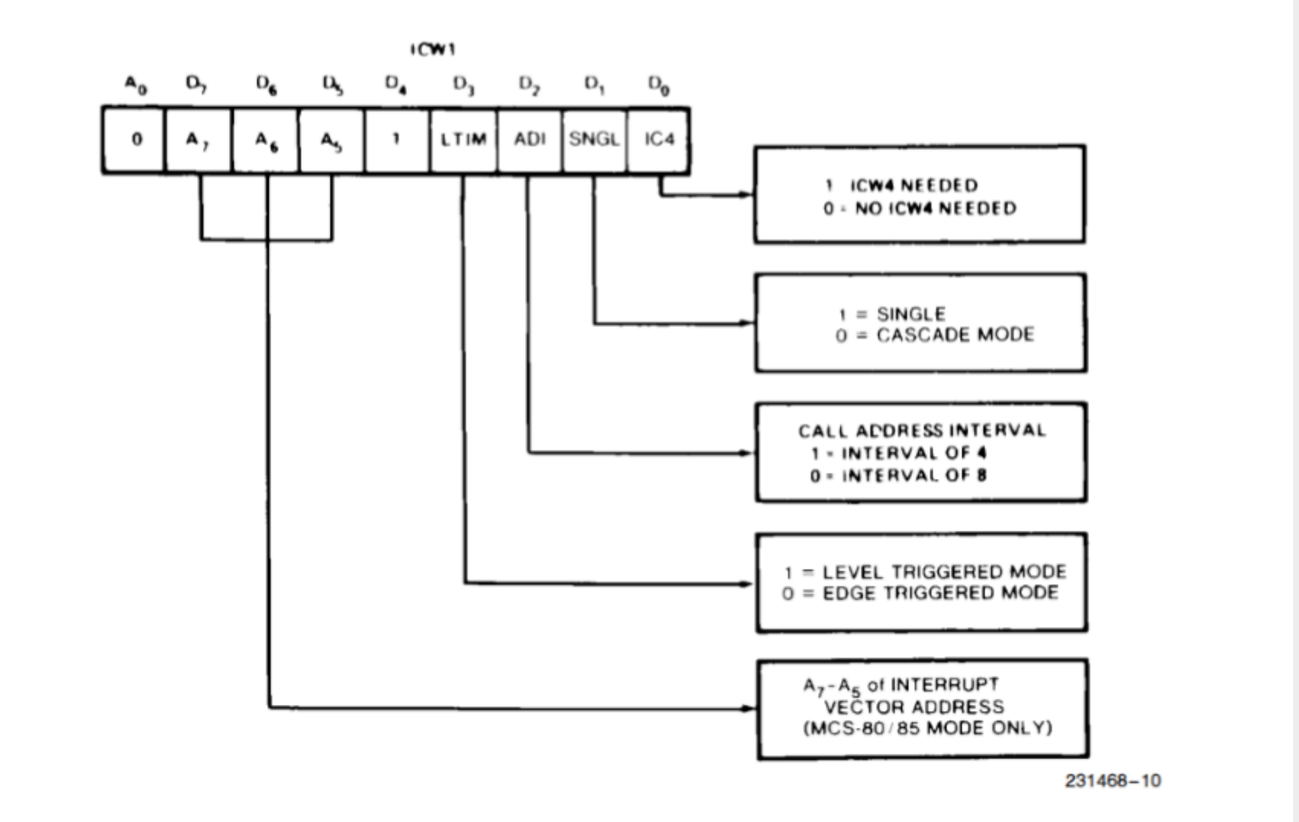
Programming the PIC in C

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

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

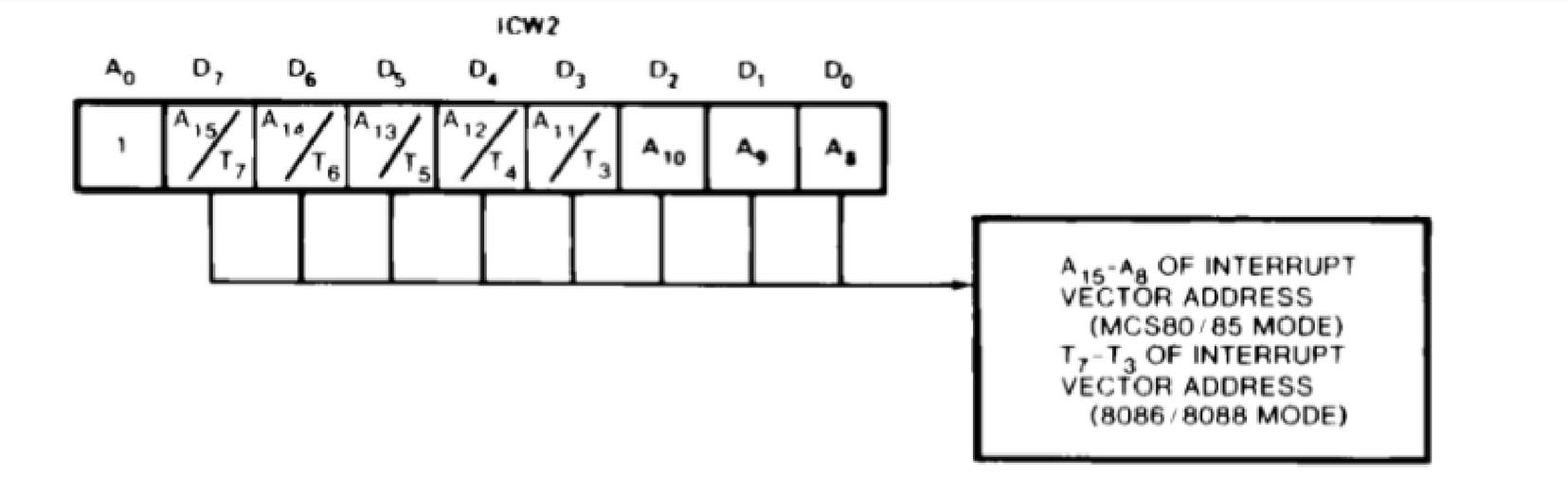
Programmable Interrupt Controller

ICW1 From the Intel Manual for x86 Architecture



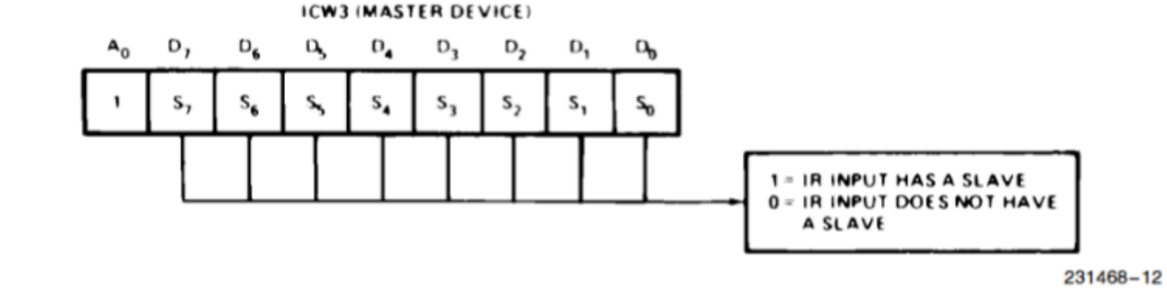
Programmable Interrupt Controller

ICW2 From the Intel Manual for x86 Architecture



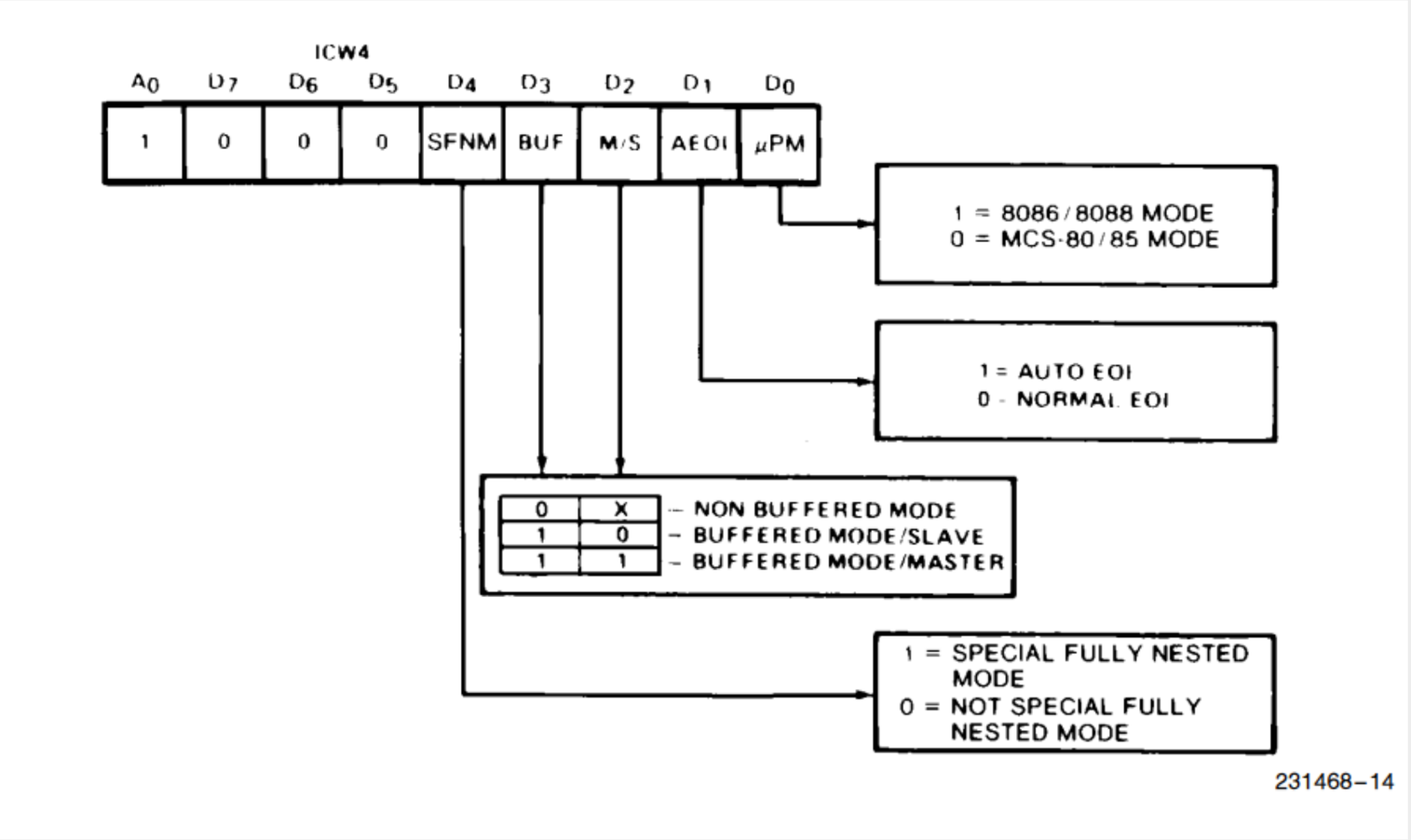
Programmable Interrupt Controller

ICW3 From the Intel Manual for x86 Architecture



Programmable Interrupt Controller

ICW4 From the Intel Manual for x86 Architecture



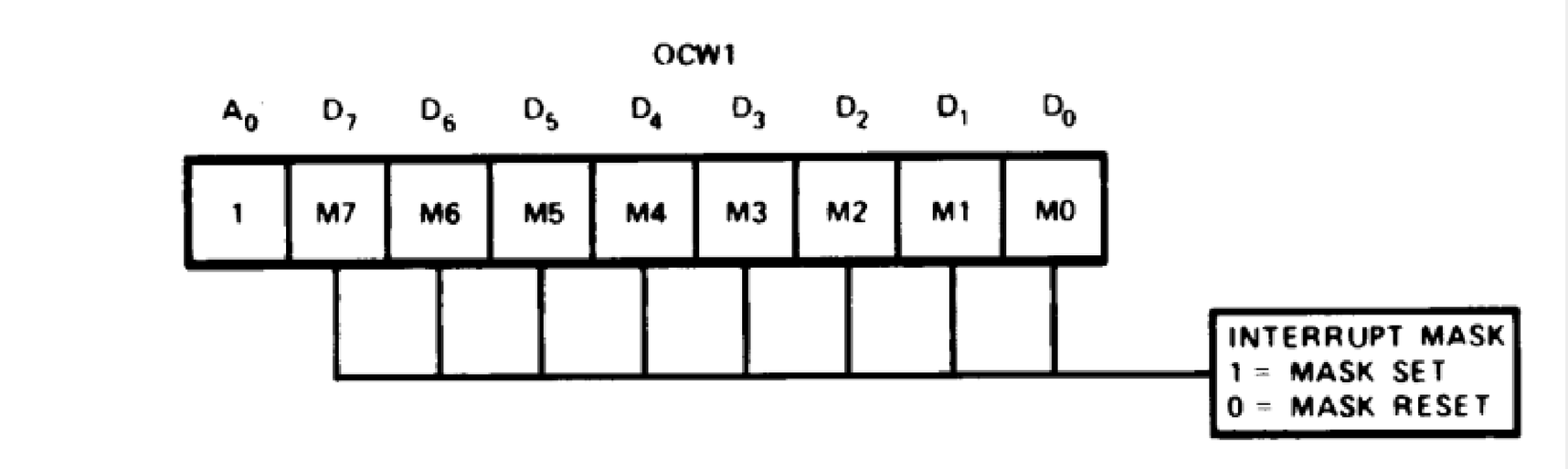
Programmable Interrupt Controller

Configuring 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();
}
```

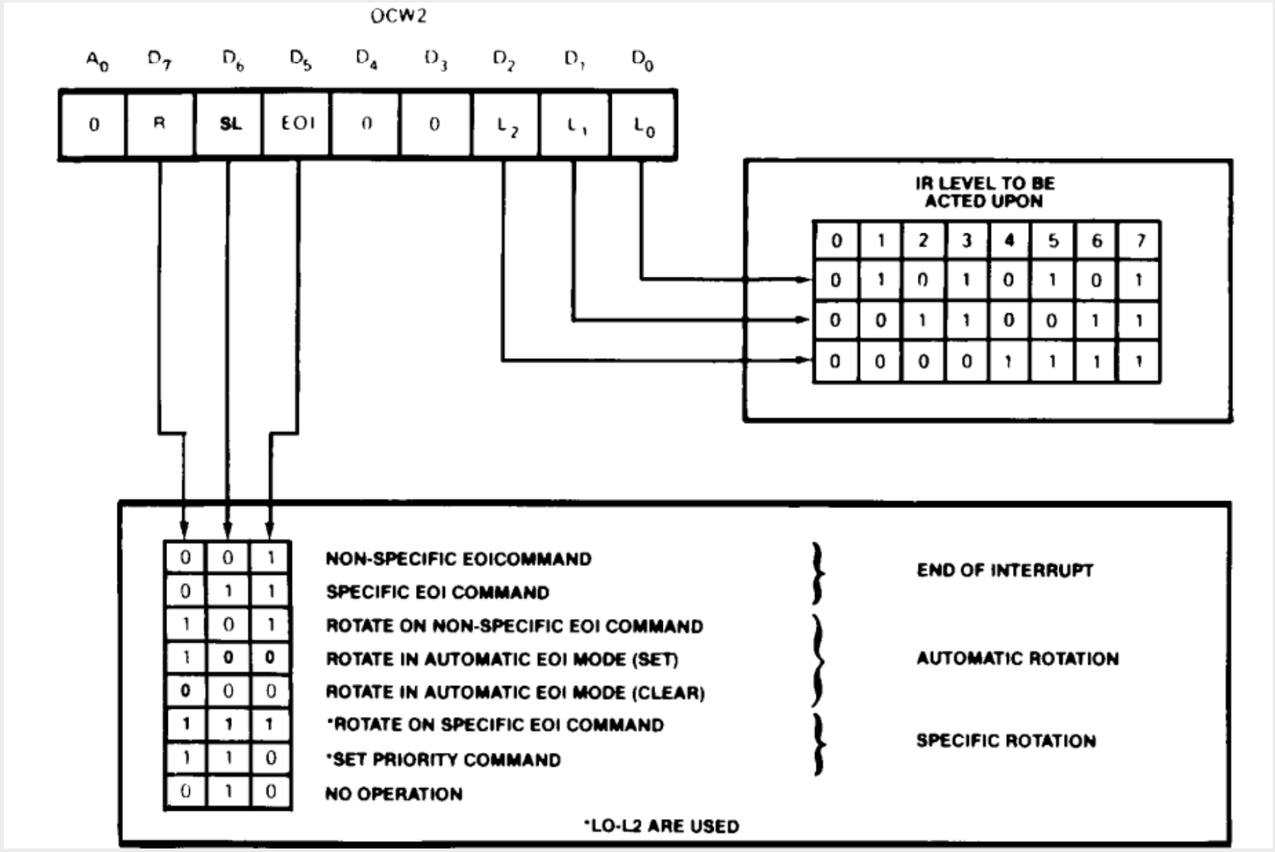
Programmable Interrupt Controller

OCW1 From the Intel Manual for x86 Architecture



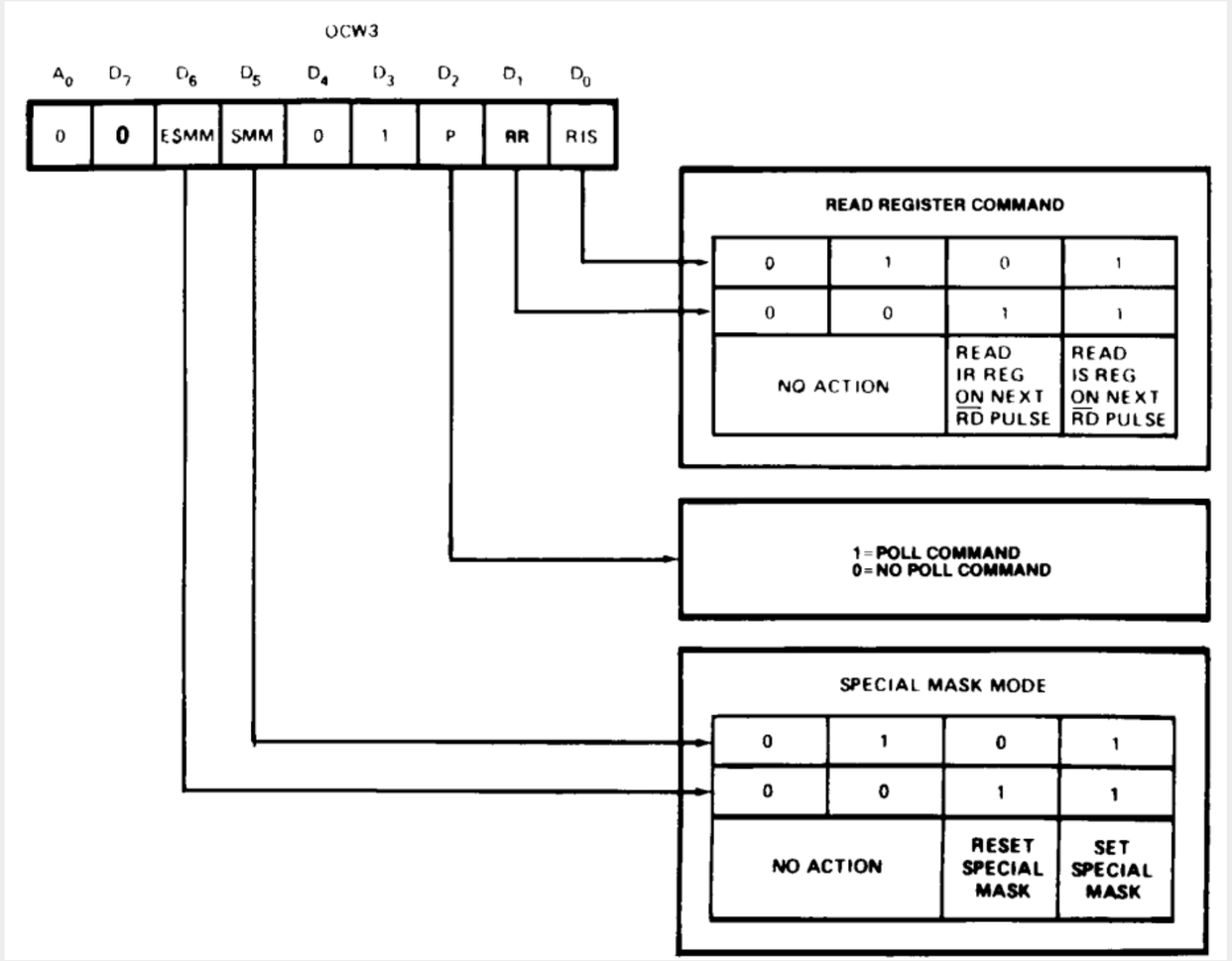
Programmable Interrupt Controller

OCW2 From the Intel Manual for x86 Architecture



Programmable Interrupt Controller

OCW3 From the Intel Manual for x86 Architecture



Programmable Interrupt Controller

OCW implementation

```
1
2  void i686_PIC_SendEndOfInterrupt(int irq)
3  {
4      if (irq >= 8)
5          i686_outb(PIC2_COMMAND_PORT, PIC_CMD_END_OF_INTERRUPT);
6      i686_outb(PIC1_COMMAND_PORT, PIC_CMD_END_OF_INTERRUPT);
7  }
8
9  void i686_PIC_Disable()
10 {
11     i686_outb(PIC1_DATA_PORT, 0xFF);        // mask all
12     i686_iowait();
13     i686_outb(PIC2_DATA_PORT, 0xFF);        // mask all
14     i686_iowait();
15 }
16
17 void i686_PIC_Mask(int irq)
18 {
19     uint8_t port;
20
21     if (irq < 8)
22     {
23         port = PIC1_DATA_PORT;
24     }
25     else
26     {
27         irq -= 8;
28         port = PIC2_DATA_PORT;
29     }
30
31     uint8_t mask = i686_inb(PIC1_DATA_PORT);
32     i686_outb(PIC1_DATA_PORT,  mask | (1 << irq));
33 }
```

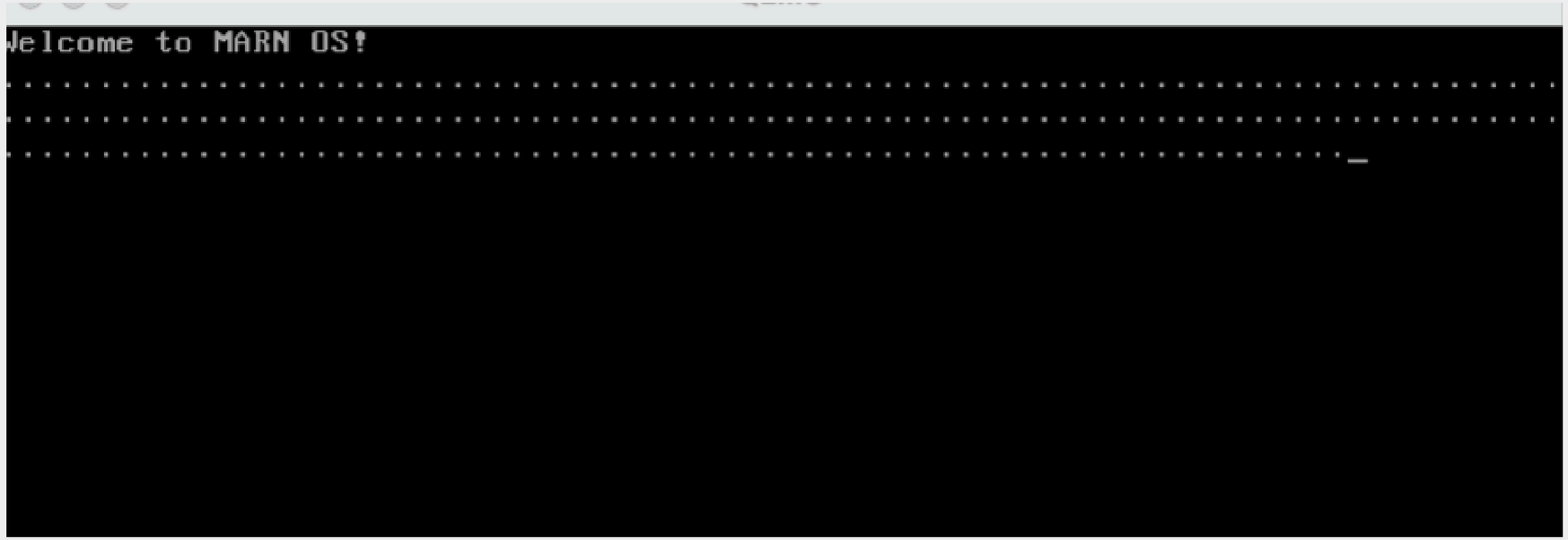
```
35 void i686_PIC_Unmask(int irq)
36 {
37     uint8_t port;
38
39     if (irq < 8)
40     {
41         port = PIC1_DATA_PORT;
42     }
43     else
44     {
45         irq -= 8;
46         port = PIC2_DATA_PORT;
47     }
48
49     uint8_t mask = i686_inb(PIC1_DATA_PORT);
50     i686_outb(PIC1_DATA_PORT,  mask & ~(1 << irq));
51 }
52
53 uint16_t i686_PIC_ReadIrqRequestRegister()
54 {
55     i686_outb(PIC1_COMMAND_PORT, PIC_CMD_READ_IRR);
56     i686_outb(PIC2_COMMAND_PORT, PIC_CMD_READ_IRR);
57     return ((uint16_t)i686_inb(PIC2_COMMAND_PORT)) | (((uint16_t)i686_inb(PIC2_COMMAND_PORT)) << 8);
58 }
59
60 uint16_t i686_PIC_ReadInServiceRegister()
61 {
62     i686_outb(PIC1_COMMAND_PORT, PIC_CMD_READ_ISR);
63     i686_outb(PIC2_COMMAND_PORT, PIC_CMD_READ_ISR);
64     return ((uint16_t)i686_inb(PIC2_COMMAND_PORT)) | (((uint16_t)i686_inb(PIC2_COMMAND_PORT)) << 8);
65 }
```

Hardware Interrupts

Hardware interrupts are signals from hardware devices. They are managed using Interrupt Requests (IRQs) numbered from 0 to 15. In the mentioned program, Timer (IRQ0) is used for timekeeping, and Keyboard Interrupt (IRQ1) captures user input like keypresses. IRQs help the operating system handle hardware events.

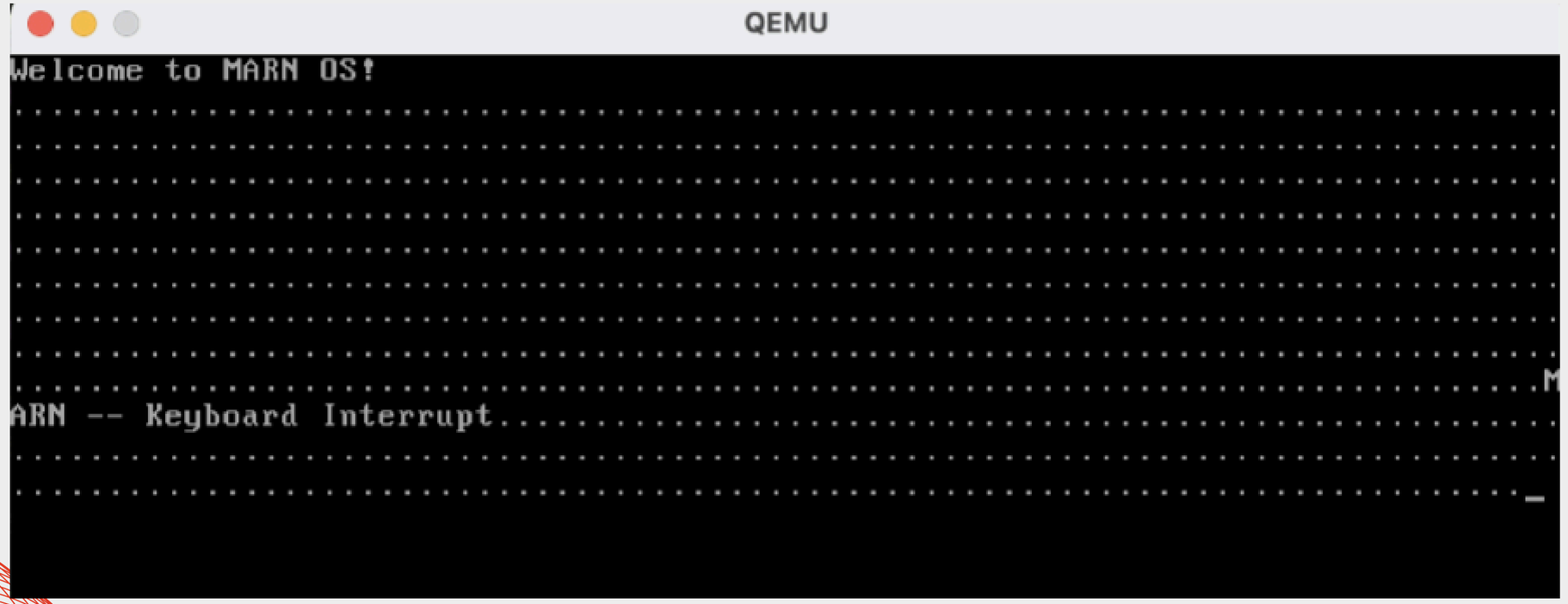
Timer : IRQ0

Handled Timer Interrupt:



Keyboard Interrupt : IRQ1

Handled Keyboard Interrupt:

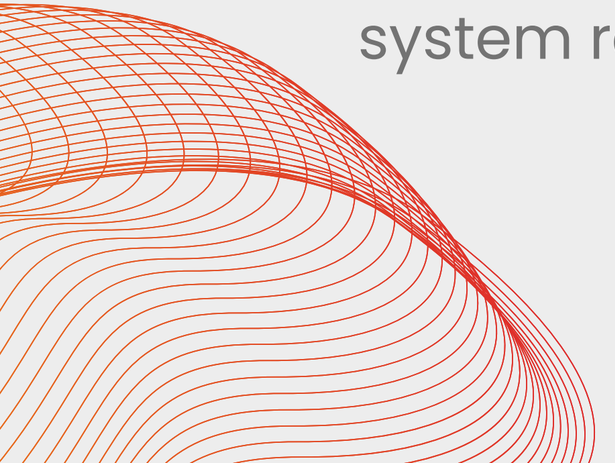


IRQ0 & IRQ1 Implementation

```
void timer(Registers* regs){
    printf(".");
    return;
}

void keyboard(Registers* regs) {
    printf("MARN -- Keyboard Interrupt");
    printf(".");
}
```

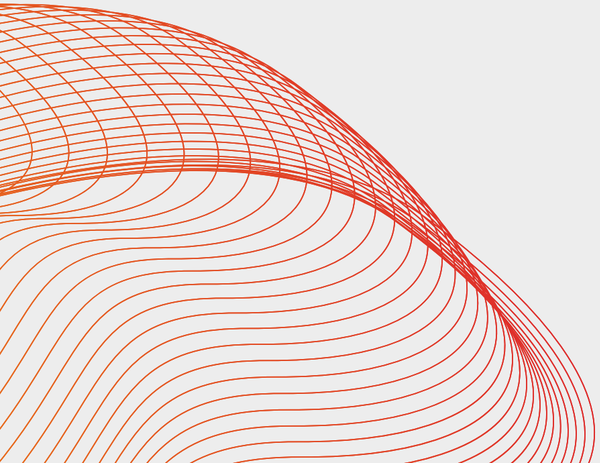
Conclusions

- Implemented advanced interrupt management features, enhancing the system's capability to both generate and handle interrupts.
- Crafted user-defined interrupts, adding a layer of customization and unhandled interrupts were now handled.
- Resolved previously unhandled interrupt scenarios, increasing system reliability



References:

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



Thank You