Lesson7 Interrupts, Keyboard and Timer

Interrupts allow the program to wait for an event trigger rather than checking continuously for an event. The event could be a keyboard key press. Once the key is pressed the program is informed by an interrupt. The interrupt would then read the key from the keyboard and place in a keyboard buffer. When the keyboard buffer is filled by pressing the enter key then a command can be processed. Polling would take up extra CPU resources where as waiting for interrupts does not take up CPU execution time.

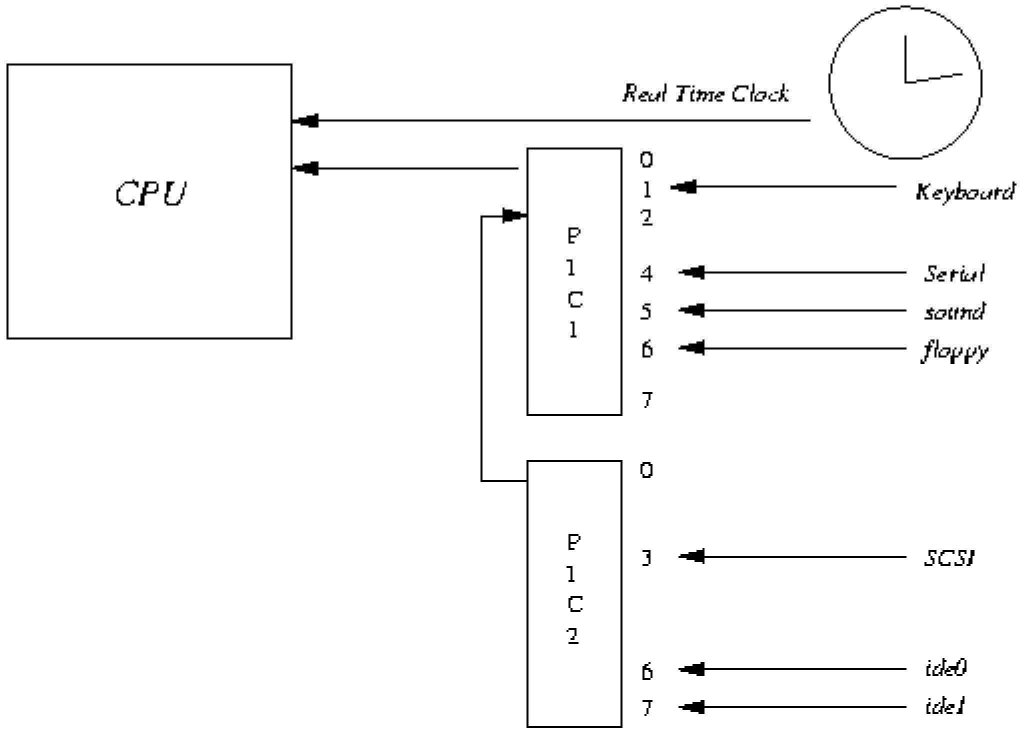
An interrupt may be invoked by hardware using an external pin on the CPU or by a software instruction int. The CPU services the interrupt by executing a subroutine called interrupt service routine (ISR). When an interrupt occurs, the processor first finishes the current instruction. It then suspends the current program and executes an ISR. The CPU pushes the value of PC address of next instruction into the stack. The CPO then loads the ISR address into PC and proceeds to execute the ISR. At the end of the ISR, the CPU pops the return address from the stack and loads it back into PC. The CPU then returns to the very next instruction in the program.

80386 Interrupt mechanism

The keyboard is connected to the computer through a physical USB When a key is pressed, the keyboard sends the key pressed to the I/O port 0x60 and sends an interrupt request IRQ 1 to the programmable interrupt controller (PIC).

The **programmable interrupt controller** (**PIC**) enables the CPU to handle interrupt requests (IRQ) coming from multiple different sources (like external I/O devices) which may occur simultaneously. It helps prioritize IRQs so that the CPU switches execution to the most appropriate interrupt handler (ISR) after the PIC assesses the IRQ's relative priorities.

16 IRQs are used. IRQs 0 to 7 are managed by one PIC, and IRQs 8 to 15 by a second PIC. The first PIC, called the master, is the only one that directly signals the CPU. The second PIC, the slave, passes the interrupt signals to the master on its IRQ 2 line, and then the master passes the signal on to the CPU. There are therefore only 15 interrupt request lines available .



Master PIC

- IRQ 0 – system timer (cannot be changed)
- IRQ 1 – keyboard on PS/2 port (cannot be changed)
- IRQ 2 – cascaded signals from IRQs 8–15
  (any devices configured to use IRQ 2 will actually be using IRQ 9)
- IRQ 3 – serial port controller for serial port 2 (shared with serial port 4, if present)
- IRQ 4 – serial port controller for serial port 1 (shared with serial port 3, if present)
- IRQ 5 – parallel port 3 or sound card
- IRQ 6 – floppy disk controller
- IRQ 7 – parallel port 1 (shared with parallel port 2, if present). It is used for printers or for any parallel port if a printer is not present. It can also be potentially be shared with a secondary sound card with careful management of the port.

Slave PIC

- IRQ 8 – real-time clock (RTC)
- IRQ 9 – Advanced Configuration and Power Interface (ACPI) system control interrupt on Intel chipsets.[3] Other chipset manufacturers might use another interrupt for this purpose, or make it available for the use of peripherals
  (any devices configured to use IRQ 2 will actually be using IRQ 9)
- IRQ 10 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or NIC)
- IRQ 11 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or NIC)
- IRQ 12 – mouse on PS/2 port
- IRQ 13 – CPU co-processor or integrated floating point unit or inter-processor interrupt (use depends on OS)
- IRQ 14 – primary ATA channel (ATA interface usually serves hard disk drives and CD drives)
- IRQ 15 – secondary ATA channel

The PIC then interrupts the CPU with a predefined interrupt number based on the external IRQ. On receiving the interrupt, the CPU will consult the interrupt descriptor table (IDT) to look up the respective interrupt handler it should invoke. After the handler has completed its task, the CPU will resume regular execution from before the interrupt.

Interrupt Descriptor Table (IDT)

The **Interrupt Descriptor Table** (**IDT**) is a data structure use to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions. An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor,

INT_NUM	Short Description **PM**
0x00	Division by zero
0x01	Single-step interrupt
0x02	NMI Non Maskable Interrupt
0x03	Breakpoint (which benefits from the shorter 0xCC encoding of INT 3)
0x04	Overflow
0x05	Bound Range Exceeded
0x06	Invalid Opcode
0x07	Coprocessor not available

0x08	Double Fault
0x09	Coprocessor Segment Overrun *(386 or earlier only)*
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Segment Fault
0x0D	General Protection Fault
0x0E	Page Fault
0x0F	*Reserved*
0x10	x87 Floating Point Exception
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception (only available with CET)

For the complete interrupt chain to work we have to setup the correct mapping inside the PIC so that our IRQs get translated to actual interrupts correctly. We must also create and load a valid IDT that contains a reference to our keyboard handler. The handler then reads all the data from the keyboard I/O port and converts it to text that we can store in a key board buffer.

IDT Structure

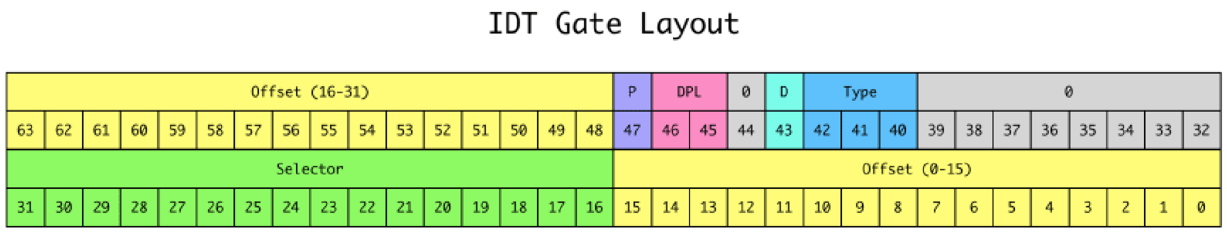
The IDT consists of 256 descriptor entries, called gates. Each of those gates is 8 bytes long and corresponds to exactly one interrupt number, determined from its position in the table. There are three types of gates: task gates, interrupt gates, and trap gates. Interrupt and trap gates can invoke custom handler functions, with interrupt gates temporarily disabling hardware interrupt handling during the handler invocation, which makes it useful for processing hardware interrupts. Task gates cause allow using the hardware task switch mechanism to pass control of the processor to another program.

We only need to define interrupt gates for now.

An interrupt gate contains the following information:

- **Offset.** The 32 bit offset represents the memory address of the interrupt handler within the respective code segment
.
- **Selector.** The 16 bit selector of the code segment to jump to when invoking the handler. This will be our kernel code segment.
- **Type.** 3 bits indicating the gate type. Will be set to 110 as we are defining an interrupt gate.
- **D.** 1 bit indicating whether the code segment is 32 bit. Will be set to 1
.
- **DPL.** 2 bits The descriptor privilege level indicates what privilege is required to invoke the handler. Will be set to 00.
- **P.** 1 bit indicating whether the gate is active. Will be set to 1.
- **0.** Some bits that always need to be set to 0 for interrupt gates.

The following diagram below illustrates the layout of an IDT gate: The IDT gate structure is 64 bits or 8 bytes in length.



To create an IDT gate in C, we first define the idt_gate_t struct type. __attribute__((packed)) tells gcc to pack the data inside the struct as tight as they are defined. Otherwise the compiler might include padding to optimize the struct layout with respect to the CPU cache size, for example.

Now we can define our IDT as an array of 256 gates and implement a setter function set_idt_gate to register a handler for interrupt n. We will make use of two small helper functions to split the 32 bit memory address of the handler.

Here are the steps to implement interrupts, keyboard and timer:

Step 1

Copy all the files from Lesson6 folder and put into Lesson 7 folder.
Make a new header file called **idt.h** and put in folder Lesson7 and put the following code in it

```
// idt.h
#pragma once

// interrupt description table setup IDT

//Segment selectors
#define KERNEL_CS 0x08

// number of IDT entries
#define IDT_ENTRIES 256

// IDT Gate structure
// How every interrupt gate (handler) is defined
#pragma pack(1)  // mingw fix for __attribute__((packed))
typedef struct {
    unsigned short low_offset; // Lower 16 bits of handler function address
    unsigned short sel; // Kernel segment selector
    unsigned char always0;
    // First byte
    // Bit 7: "Interrupt is present"
    // Bits 6-5: Privilege level of caller (0=kernel..3=user)
    // Bit 4: Set to 0 for interrupt gates
    // Bits 3-0: bits 1110 = decimal 14 = "32 bit interrupt gate"
    unsigned char flags;
    unsigned short high_offset; // Higher 16 bits of handler function address
} __attribute__((packed)) idt_gate_t;
// A pointer to the array of interrupt handlers.
// Assembly instruction 'lidt' will read it
#pragma pack(1)  // mingw fix for __attribute__((packed))
typedef struct {
    unsigned short limit;
    unsigned int base;
} __attribute__((packed)) idt_register_t;

// register an interrupt handlet
void set_idt_gate(int n, unsigned int handler);

//  our IDT as an array of 256 gates
void load_idt();
```

Step 2

Make a companion implementation code file called **idt.c** and put in a folder Lesson7 and put the following code in it.

```
// idt.c

// interrupt description table setup IDT

#include "idt.h"
#include "strings.h"

idt_gate_t idt[IDT_ENTRIES]; // idt table
idt_register_t idt_reg; // idt register

// register interrupt handler
void set_idt_gate(int n, unsigned int handler) {
    idt[n].low_offset = low_16(handler);
    idt[n].sel = KERNEL_CS;
    idt[n].always0 = 0;
    idt[n].flags = 0x8E;
    idt[n].high_offset = high_16(handler);
}

// load idt register
void load_idt() {
    idt_reg.base = (unsigned int)&idt;
    idt_reg.limit = IDT_ENTRIES * sizeof(idt_gate_t) - 1;
    // load &idt_reg using lidt instruction
    asm volatile("lidt (%0)" : : "r" (&idt_reg));
}
```

Setting Up the Internal ISRs

An interrupt handler is also referred to as a interrupt service routines (ISR). The first 32 ISRs are reserved for CPU specific interrupts, such as exceptions and faults. The next 32 ISRs will be mapped to our IRQ service routines.

Setting these up is crucial as they are the only way for us to know if we are doing something wrong when remapping the PIC and defining the IRQs later.

First, we define a generic ISR handler function in C. It can extract all necessary information related to the interrupt and act accordingly. For now we will have a simple lookup array that contains a string representation for each interrupt number.

To make sure we have all information available, we are going to pass a struct of type registers_t to the function that is defined as follows:

The reason this struct is so complex lies in the fact that we are going to invoke the handler function (which is written in C) from within assembly.

Before a function is invoked, C expects the arguments to be present on the stack. The stack will contain some information already and we are extending it with additional information.

Step 3

Make a new header file called isr.h and put the following code in it.

```
// isr.h
#pragma once

// setup interrupt service routines for ISR's
// setup interrupt request routines for IRQ's

// ISRs reserved for CPU exceptions
extern void isr0();
extern void isr1();
extern void isr2();
extern void isr3();
extern void isr4();
extern void isr5();
extern void isr6();
extern void isr7();
extern void isr8();
extern void isr9();
extern void isr10();
extern void isr11();
extern void isr12();
extern void isr13();
extern void isr14();
extern void isr15();
extern void isr16();
extern void isr17();
extern void isr18();
extern void isr19();
extern void isr20();
extern void isr21();
extern void isr22();
extern void isr23();
extern void isr24();
extern void isr25();
extern void isr26();
extern void isr27();
extern void isr28();
extern void isr29();
extern void isr30();
extern void isr31();

// IRQ definitions
extern void irq0();
extern void irq1();
extern void irq2();
extern void irq3();
extern void irq4();
```

```c
extern void irq5();
extern void irq6();
extern void irq7();
extern void irq8();
extern void irq9();
extern void irq10();
extern void irq11();
extern void irq12();
extern void irq13();
extern void irq14();
extern void irq15();

// IRQ numbers
#define IRQ0 32
#define IRQ1 33
#define IRQ2 34
#define IRQ3 35
#define IRQ4 36
#define IRQ5 37
#define IRQ6 38
#define IRQ7 39
#define IRQ8 40
#define IRQ9 41
#define IRQ10 42
#define IRQ11 43
#define IRQ12 44
#define IRQ13 45
#define IRQ14 46
#define IRQ15 47

// registers str tructure which aggregates many registers.
// It matches exactly the pushes on interrupt.asm. From the bottom:
// Pushed by the processor automatically
// push byte`s on the isr-specific code: error code, then int number
// All the registers by pusha
// push eax` whose lower 16-bits contain DS
typedef struct {
    unsigned int ds; // Data segment selector
    unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax; // Pushed by pusha.
    unsigned int int_no, err_code; // Interrupt number and error code
    unsigned int eip, cs, eflags, useresp, ss; // Pushed by the processor automatically
} registers_t;

// install isr's
void isr_install();

// handle isr request
void isr_handler(registers_t* r);

// handle irq request
void irq_handler(registers_t* r);

// isr function pointer
typedef void (*isr_t)(registers_t*);

// register interrupt handler
void register_interrupt_handler(unsigned char n, isr_t handler);
```

Step 4

Make a C companion implementation code file called isr.c

```
// isr.c
// interrupt service routines
#include "isr.h"
#include "idt.h"
#include "screen.h"
#include "ports.h"
#include "strings.h"

// table of ibterupt handlers
isr_t interrupt_handlers[256];

// install isrs individually
// isr's names must match with interrupts.asm
void isr_install() {

    // Install the ISRs 0-32 individually
    // isr's names must match with interrupts.asm
    set_idt_gate(0, (unsigned int) isr0);
    set_idt_gate(1, (unsigned int) isr1);
    set_idt_gate(2, (unsigned int) isr2);
    set_idt_gate(3, (unsigned int) isr3);
    set_idt_gate(4, (unsigned int) isr4);
    set_idt_gate(5, (unsigned int) isr5);
    set_idt_gate(6, (unsigned int) isr6);
    set_idt_gate(7, (unsigned int) isr7);
    set_idt_gate(8, (unsigned int) isr8);
    set_idt_gate(9, (unsigned int) isr9);
    set_idt_gate(10, (unsigned int) isr10);
    set_idt_gate(11, (unsigned int) isr11);
    set_idt_gate(12, (unsigned int) isr12);
    set_idt_gate(13, (unsigned int) isr13);
    set_idt_gate(14, (unsigned int) isr14);
    set_idt_gate(15, (unsigned int) isr15);
    set_idt_gate(16, (unsigned int) isr16);
    set_idt_gate(17, (unsigned int) isr17);
    set_idt_gate(18, (unsigned int) isr18);
    set_idt_gate(19, (unsigned int) isr19);
    set_idt_gate(20, (unsigned int) isr20);
    set_idt_gate(21, (unsigned int) isr21);
    set_idt_gate(22, (unsigned int) isr22);
    set_idt_gate(23, (unsigned int) isr23);
    set_idt_gate(24, (unsigned int) isr24);
    set_idt_gate(25, (unsigned int) isr25);
    set_idt_gate(26, (unsigned int) isr26);
    set_idt_gate(27, (unsigned int) isr27);
    set_idt_gate(28, (unsigned int) isr28);
    set_idt_gate(29, (unsigned int) isr29);
    set_idt_gate(30, (unsigned int) isr30);
    set_idt_gate(31, (unsigned int) isr31);
    // Remap the PIC
    // ICW1
    port_byte_out(0x20, 0x11);
    port_byte_out(0xA0, 0x11);

    // ICW2
    port_byte_out(0x21, 0x20);
    port_byte_out(0xA1, 0x28);

    // ICW3
    port_byte_out(0x21, 0x04);
    port_byte_out(0xA1, 0x02);

    // ICW4
    port_byte_out(0x21, 0x01);
    port_byte_out(0xA1, 0x01);

    // OCW1
    port_byte_out(0x21, 0x0);
    port_byte_out(0xA1, 0x0);

    // Install the IRQs ISRs 32-47 individually
    // irq's names must match with interrupts.asm
    set_idt_gate(32, (unsigned int)irq0);
    set_idt_gate(33, (unsigned int)irq1);
    set_idt_gate(34, (unsigned int)irq2);
    set_idt_gate(35, (unsigned int)irq3);
    set_idt_gate(36, (unsigned int)irq4);
    set_idt_gate(37, (unsigned int)irq5);
    set_idt_gate(38, (unsigned int)irq6);
    set_idt_gate(39, (unsigned int)irq7);
    set_idt_gate(40, (unsigned int)irq8);
    set_idt_gate(41, (unsigned int)irq9);
    set_idt_gate(42, (unsigned int)irq10);
    set_idt_gate(43, (unsigned int)irq11);
    set_idt_gate(44, (unsigned int)irq12);
    set_idt_gate(45, (unsigned int)irq13);
    set_idt_gate(46, (unsigned int)irq14);
    set_idt_gate(47, (unsigned int)irq15);

    load_idt(); // Load with ASM
}

// handle isr request
void isr_handler(registers_t* r) {

    // isr exception messages
    char msg0[] = "Division By Zero";
    char msg1[] = "Debug";
    char msg2[] = "Non Maskable Interrupt";
    char msg3[] = "Breakpoint";
    char msg4[] = "Into Detected Overflow";
    char msg5[] = "Out of Bounds";
    char msg6[] = "Invalid Opcode";
    char msg7[] = "No Coprocessor";
    char msg8[] = "Double Fault";
    char msg9[] = "Coprocessor Segment Overrun";
    char msg10[] = "Bad TSS";
    char msg11[] = "Segment Not Present";
    char msg12[] = "Stack Fault";
    char msg13[] = "General Protection Fault";
    char msg14[] = "Page Fault";
    char msg15[] = "Unknown Interrupt";
    char msg16[] = "Coprocessor Fault";
    char msg17[] = "Alignment Check";
    char msg18[] = "Machine Check";
    char msg19[] = "Reserved";
    char msg20[] = "Reserved";
    char msg21[] = "Reserved";
    char msg22[] = "Reserved";
    char msg23[] = "Reserved";
    char msg24[] = "Reserved";
    char msg25[] = "Reserved";
    char msg26[] = "Reserved";
    char msg27[] = "Reserved";
    char msg28[] = "Reserved";
    char msg29[] = "Reserved";
    char msg30[] = "Reserved";
    char msg31[] = "Reserved";

    // array of message pointers
    char* messages[] = { msg0, msg1, msg2, msg3, msg4, msg5, msg6, msg7, msg8, msg9,
    msg10, msg11,msg12, msg13, msg14, msg15, msg16, msg17 ,msg18, msg19, msg20, msg21,
    msg22, msg23, msg24, msg25, msg26, msg27, msg28, msg29, msg30, msg31};

    // report isr interrupt
    char msg[] = "received ISR interrupt: ";
    print_string(msg);
    char s[32];

    // print interrupt number
    int_to_string(r->int_no, s);
    print_string(s);
    print_nl();

    // print interrupt message
    print_string(messages[r->int_no]);
    print_nl();
}

// register isr handler
void register_interrupt_handler(unsigned char n, isr_t handler) {
    interrupt_handlers[n] = handler;
}
```

```
// irq handler
void irq_handler(registers_t *r) {

    //char msg[] = "received IRQ interrupt: ";
    //print_string(msg);

    // Handle the interrupt
    if (interrupt_handlers[r->int_no] !=0 && r->int_no > 32) {

        //char text[20];
        //int_to_string(r->int_no, text);
        //print_string(text);

        isr_t handler = interrupt_handlers[r->int_no];
        handler(r);
    }

    // EOI
    // tell PIC we are finished
    if (r->int_no >= 40) {
        port_byte_out(0xA0, 0x20); /* follower */
    }
    port_byte_out(0x20, 0x20); /* leader */
}
```

Define ISR's in Assembly language

We use assembly language code that defines the first 32 ISRs. Unfortunately there is no way to know which gate was used to invoke the handler so we need one handler for each gate. We have to define the labels as global so that we can reference them from our C code later.

Step 5

Make a new file called interrupts.asm and put the following code in it.

```
; interrupts.asm
; Defined in isr.c
[extern _isr_handler]
[extern _irq_handler]

; Common ISR code
isr_common_stub:
    ; 1. Save CPU state
       pusha ; Pushes edi,esi,ebp,esp,ebx,edx,ecx,eax
       mov ax, ds ; Lower 16-bits of eax = ds.
       push eax ; save the data segment descriptor
       mov ax, 0x10  ; kernel data segment descriptor
       mov ds, ax
       mov es, ax
       mov fs, ax
       mov gs, ax
    ; 2. Call C handler
    push esp ; push registers_t *r pointer
       call _isr_handler
       pop eax ; clear pointer afterwards

    ; 3. Restore state
       pop eax
       mov ds, ax
       mov es, ax
       mov fs, ax
       mov gs, ax
       popa
       add esp, 8 ; Cleans up the pushed error code and pushed ISR number
       iret ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP

; Common IRQ code. Identical to ISR code except for the 'call'
; and the 'pop ebx'
irq_common_stub:
    ; 1. Save CPU state
    pusha
    mov ax, ds
    push eax
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; 2. Call C handler
    push esp
    call _irq_handler ; Different than the ISR code
    pop ebx  ; Different than the ISR code

    ; 3. Restore state
    pop ebx
    mov ds, bx
    mov es, bx
    mov fs, bx
    mov gs, bx
    popa
    add esp, 8
    iret

; We don't get information about which interrupt was caller
; when the handler is run, so we will need to have a different handler
; for every interrupt.
; Furthermore, some interrupts push an error code onto the stack but others
; don't, so we will push a dummy error code for those which don't, so that
; we have a consistent stack for all of them.
; First make the ISRs global
global _isr0
global _isr1
global _isr2
global _isr3
global _isr4
global _isr5
global _isr6
global _isr7
global _isr8
global _isr9
global _isr10
global _isr11
global _isr12
global _isr13
global _isr14
global _isr15
global _isr16
global _isr17
global _isr18
global _isr19
global _isr20
global _isr21
global _isr22
global _isr23
global _isr24
global _isr25
global _isr26
global _isr27
global _isr28
global _isr29
global _isr30
global _isr31

; 0: Divide By Zero Exception
_isr0:
    push byte 0
    push byte 0
    jmp isr_common_stub

; 1: Debug Exception
_isr1:
    push byte 0
    push byte 1
    jmp isr_common_stub

; 2: Non Maskable Interrupt Exception
_isr2:
    push byte 0
    push byte 2
    jmp isr_common_stub
; 3: Int 3 Exception
_isr3:
    push byte 0
    push byte 3
    jmp isr_common_stub
```

```
; 4: INTO Exception
_isr4:
    push byte 0
    push byte 4
    jmp isr_common_stub

; 5: Out of Bounds Exception
_isr5:
    push byte 0
    push byte 5
    jmp isr_common_stub

; 6: Invalid Opcode Exception
_isr6:
    push byte 0
    push byte 6
    jmp isr_common_stub

; 7: Coprocessor Not Available Exception
_isr7:
    push byte 0
    push byte 7
    jmp isr_common_stub

; 8: Double Fault Exception (With Error Code!)
_isr8:
    push byte 8
    jmp isr_common_stub

; 9: Coprocessor Segment Overrun Exception
_isr9:
    push byte 0
    push byte 9
    jmp isr_common_stub

; 10: Bad TSS Exception (With Error Code!)
_isr10:
    push byte 10
    jmp isr_common_stub

; 11: Segment Not Present Exception (With Error Code!)
_isr11:
    push byte 11
    jmp isr_common_stub

; 12: Stack Fault Exception (With Error Code!)
_isr12:
    push byte 12
    jmp isr_common_stub

; 13: General Protection Fault Exception (With Error Code!)
_isr13:
    push byte 13
    jmp isr_common_stub
```

```
; 14: Page Fault Exception (With Error Code!)
_isr14:
    push byte 14
    jmp isr_common_stub

; 15: Reserved Exception
_isr15:
    push byte 0
    push byte 15
    jmp isr_common_stub

; 16: Floating Point Exception
_isr16:
    push byte 0
    push byte 16
    jmp isr_common_stub

; 17: Alignment Check Exception
_isr17:
    push byte 0
    push byte 17
    jmp isr_common_stub

; 18: Machine Check Exception
_isr18:
    push byte 0
    push byte 18
    jmp isr_common_stub

; 19: Reserved
_isr19:
    push byte 0
    push byte 19
    jmp isr_common_stub

; 20: Reserved
_isr20:
    push byte 0
    push byte 20
    jmp isr_common_stub

; 21: Reserved
_isr21:
    push byte 0
    push byte 21
    jmp isr_common_stub

; 22: Reserved
_isr22:
    push byte 0
    push byte 22
    jmp isr_common_stub
; 23: Reserved

_isr23:
    push byte 0
    push byte 23
    jmp isr_common_stub
```

```
; 24: Reserved
_isr24:
    push byte 0
    push byte 24
    jmp isr_common_stub

; 25: Reserved
_isr25:
    push byte 0
    push byte 25
    jmp isr_common_stub

; 26: Reserved
_isr26:
    push byte 0
    push byte 26
    jmp isr_common_stub

; 27: Reserved
_isr27:
    push byte 0
    push byte 27
    jmp isr_common_stub

; 28: Reserved
_isr28:
    push byte 0
    push byte 28
    jmp isr_common_stub

; 29: Reserved
_isr29:
    push byte 0
    push byte 29
    jmp isr_common_stub

; 30: Reserved
_isr30:
    push byte 0
    push byte 30
    jmp isr_common_stub

; 31: Reserved
_isr31:
    push byte 0
    push byte 31
    jmp isr_common_stub

; IRQs
global _irq0
global _irq1
global _irq2
global _irq3
global _irq4
global _irq5
global _irq6
global _irq7
global _irq8
global _irq9
global _irq10
global _irq11
global _irq12
global _irq13
global _irq14
global _irq15

; IRQ handlers
_irq0:
       push byte 0
       push byte 32
       jmp irq_common_stub

_irq1:
       push byte 1
       push byte 33
       jmp irq_common_stub

_irq2:
       push byte 2
       push byte 34
       jmp irq_common_stub

_irq3:
       push byte 3
       push byte 35
       jmp irq_common_stub

_irq4:
       push byte 4
       push byte 36
       jmp irq_common_stub

_irq5:
       push byte 5
       push byte 37
       jmp irq_common_stub

_irq6:
       push byte 6
       push byte 38
       jmp irq_common_stub

_irq7:
       push byte 7
       push byte 39
       jmp irq_common_stub
_irq8:
       push byte 8
       push byte 40
       jmp irq_common_stub

_irq9:
       push byte 9
       push byte 41
       jmp irq_common_stub
```

```
_irq10:
       push byte 10
       push byte 42
       jmp irq_common_stub

_irq11:
       push byte 11
       push byte 43
       jmp irq_common_stub

_irq12:
       push byte 12
       push byte 44
       jmp irq_common_stub

_irq13:
       push byte 13
       push byte 45
       jmp irq_common_stub

_irq14:
       push byte 14
       push byte 46
       jmp irq_common_stub

_irq15:
       push byte 15
       push byte 47
       jmp irq_common_stub
```

ISR – ISQ Code explanation

Note: all assembly language labels id's start with a underscore _. This is needed so that the C language code can recognize the assembly language labels when everything get liked together (_ is only used with the mingw C compiler).

Each procedure makes sure that the int_no and err_code are on the stack before handing over to the common ISR procedure, which we will look at in a moment. The first push (err_code), if present, represents error information that is specific to certain exceptions like stack faults. If such an exception occurs, the CPU will push this error information to the stack for us. To have a consistent stack for all ISRs, we are pushing a 0 byte in the cases where no error information is available. The second push corresponds to the interrupt number

The common ISR procedure will fill the stack with all information required for registers_t, prepare the segment pointers to invoke our kernel ISR handler isr_handler, push the stack pointer (which is a pointer to registers_t actually) to the stack, call isr_handler, and clean up afterwards so that the CPU can resume where it was interrupted. isr_handler has to be marked as extern, because it will be defined in C.

We can register the first 32 ISRs in our IDT using the set_idt_gate function inside the following isr_install function inside the isr.c

```
void isr_install() {
   // handle isr's
    set_idt_gate(0, (uint32_t) isr0);
   set_idt_gate(1, (uint32_t) isr1);
    //  isr2 to isr39 goes here
   set_idt_gate(31, (uint32_t) isr31);

   // rest of code goes here
}
```

Now that we have the CPU internal interrupt handlers in place, we can now remap the PIC and set up the IRQ handlers.

Remapping the PIC

The 8259 PIC is responsible for managing hardware interrupts. We will utilize a cascade of two PICs, whereas each of them can handle 8 different IRQs. The secondary chip is connected to the primary chip through IRQ 2, effectively giving us 15 different IRQs to handle.

On startup the BIOS previously programs the PIC with default values for the 16 bit real mode, where the first 8 IRQs are mapped to the first 8 gates in the IDT. In protected mode however, these conflict with the first 32 gates that are reserved for CPU internal interrupts. Thus, we need to reprogram (remap) the PIC to avoid conflicts.

Programming the PIC can be done by accessing the respective I/O ports. The primary PIC uses ports 0x20 (command) and 0x21 (data). The secondary PIC uses ports 0xA0 (command) and 0xA1 (data). The programming happens by sending four initialization command words (ICWs).

First, we have to send the initialize command ICW1 (0x11) to both PICs. They will then wait for the following three inputs on the data ports:

- ICW2 (IDT offset). Will be set to 0x20 (32) for the primary and 0x28 (40) for the secondary PIC.
- ICW3 (wiring between PICs). We will tell the primary PIC to accept IRQs from the secondary PIC on IRQ 2 (0x04, which is 0b00000100). The secondary PIC will be marked as secondary by setting 0x02 = 0b00000010.
- ICW4 (mode). We set 0x01 = 0b00000001 in order to enable 8086 mode.

We finally send the first operational command word
(OCW1) 0x00 = 0b00000000 to enable all IRQs (no masking).

Using the port_byte_out function from ports.c we can extend isr_install to perform the PIC remapping as follows in isr.c file

```
void isr_install() {
   //  set internal ISRs
   //  remap PIC

   // ICW1
   port_byte_out(0x20, 0x11);
   port_byte_out(0xA0, 0x11);

  // ICW2
   port_byte_out(0x21, 0x20);
   port_byte_out(0xA1, 0x28);

   // ICW3
   port_byte_out(0x21, 0x04);
   port_byte_out(0xA1, 0x02);

   // ICW4
   port_byte_out(0x21, 0x01);
   port_byte_out(0xA1, 0x01);

   // OCW1
   port_byte_out(0x21, 0x0);
    port_byte_out(0xA1, 0x0);

    // rest of code goes here
}
```

Now that we successfully remapped the PIC to send IRQs to the interrupt gates 32-47 we can register the respective ISRs.

Setting Up IRQ ISR Handlers

Adding the ISRs to handle IRQs is very similar to the first 32 CPU internal ISRs we created. First, we extend the IDT by adding gates for our IRQs 0-15 in file isr.c

```
void isr_install() {

   // internal ISRs code
   //  remap PIC code

    // handle IRQ’s ISRs
   // IRQ ISRs (primary PIC)
   set_idt_gate(32, (uint32_t)irq0);

   //    set_idt_gate(32, (uint32_t)irq1) to  set_idt_gate(32, (uint32_t)irq6)
   set_idt_gate(39, (uint32_t)irq7);

  // IRQ ISRs (secondary PIC)
   set_idt_gate(40, (uint32_t)irq8);

   // set_idt_gate(40, (uint32_t)irq9) to set_idt_gate(40, (uint32_t)irq14);

   set_idt_gate(47, (uint32_t)irq15);
}
```

Now we add the IRQ procedure labels to our assembler code. We are pushing the IRQ number as well as the interrupt number to the stack before calling the irq_common_stub in file assembler.asm

```
global irq0

;  global irq1  to global irq14

global irq15

irq0:
   push byte 0
   push byte 32
   jmp irq_common_stub
```

```
; irq1 to irq14 go here

irq15:
   push byte 15
   push byte 47
   jmp irq_common_stub
```

The irq_common_stub is defined analogous to the isr_common_stub and it will call a C the function irq_handler. The IRQ handler will be defined a bit more modular though, as we want to be able to add individual handlers dynamically when loading the kernel, such as our keyboard handler. To do that we initialize an array of interrupt handlers isr_t which are functions that take the previously defined registers_t in file isr.c

```
// array of interrupt handlers isr_t
typedef void (*isr_t)(registers_t *);

isr_t interrupt_handlers[256];
```

The irq_handler will retrieve the respective handler from the interrupt_handlers array based on the interrupt number and invoke it with the given registers_t. Note that due to the PIC protocol we must send an end of interrupt (EOI) command to the involved PICs (only primary for IRQ 0-7, both PIC's for IRQ 8-15). This is required for the PIC to know that the interrupt is handled and it can send further interrupts in file isr.c.

```
void irq_handler(registers_t *r) {
   if (interrupt_handlers[r->int_no] != 0) {
      isr_t handler = interrupt_handlers[r->int_no];
      handler(r);
   }

   port_byte_out(0x20, 0x20); // primary EOI
   if (r->int_no < 40) {
      port_byte_out(0xA0, 0x20); // secondary EOI
   }
}
```

The IDT is defined and we only need to tell the CPU to load it.

Loading the IDT

The IDT can be loaded using the lidt instruction. To be precise, lidt does not load the IDT but instead an IDT descriptor. The IDT descriptor contains the size (limit in bytes) and the base address of the IDT. We can model the descriptor as a struct like so in file idt.c

```
#pragma pack(1)
typedef struct {
   uint16_t limit;
   uint32_t base;
} __attribute__((packed)) idt_register_t;
```

We can then call lidt inside a function called load_idt. It sets the base by obtaining the pointer to the idt gate array and computes the memory limit by multiplying the number of IDT gates (256) with the size of each gate. As usual, the limit is the size – 1 in file idt.c

```
idt_register_t idt_reg;

void load_idt() {
   idt_reg.base = (uint32_t) &idt;
   idt_reg.limit = IDT_ENTRIES * sizeof(idt_gate_t) - 1;
   asm volatile("lidt (%0)" : : "r" (&idt_reg));
}
```

Here is the final modification of our isr_install function, loading the IDT after we installed all ISRs in file idt.c.

```
void isr_install() {

  // internal ISRs code
   // PIC remapping code
   // IRQ ISRs code

   load_idt();
}
```

Keyboard Handler

When a key is pressed, and isr1 interrupt is generated and the key is stored in a key buffer. We need a way to identify which key was pressed. This can be done by reading the scan code of the respective keys. Our keyboard reader handles upper case, lower case , shift and caps lock. shift and caps lock are stored in a flag because these are separate scan codes. We have separate tables for keys and shift keys. Keys are store in a key buffer. When the enter button is pressed the **execute_command(key_buffer**) function is called to send the contents of key buffer to our command procesor. When the backspace button is pressed the last character is taken out of the key buffer.

Step 6
Make a new header file called keyboard.h and put the following code in it

```
// keyboard.h
#pragma once

// keyboard constants
#define BACKSPACE 0x08
#define ENTER 0x0a
#define E0_SIGN (1 << 0)
#define SHIFT (1 << 1)
#define CAPS_LOCK (1 << 2)

// keyboard functions

// initialize keyboard
void init_keyboard();
```

Step 7
Make the C companion implantation code file called keyboard.c:

```
// keyboard.c
// handle keyboard
#include "keyboard.h"
#include "screen.h"
#include "strings.h"
#include "isr.h"
#include "ports.h"
#include "kernel.h"


// keyboard flag
static unsigned int keyflag;
```

```c
// key board keys get put here
static char key_buffer[256] = "";


// this function is called when a key is pressed
static void keyboard_callback(registers_t* regs) {


    // shift codes
    unsigned char shift_code[256] = {
    [0x2A] = SHIFT,[0x36] = SHIFT,[0xAA] = SHIFT,[0xB6] = SHIFT
    };

    // Caps Lock codes
    unsigned char lock_code[256] = {
        [0x3A] = CAPS_LOCK
    };

    // keys
    char key_map[256] = {
    0, 0, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0',
    '-', '=', '\b', 0, 'q', 'w', 'e', 'r', 't', 'y', 'u',
    'i', 'o', 'p', '[', ']', '\n', 0, 'a', 's', 'd', 'f',
    'g', 'h', 'j', 'k', 'l', ';', '\'', '`', 0, '\\', 'z',
    'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0, '*', 0, ' '
    };

    // shift keys
    char shift_key_map[256] = {
        0, 1, '!', '@', '#', '$', '%', '^', '&', '*', '(', ')',
        '_', '+', '\b', '\t', 'Q', 'W', 'E', 'R', 'T', 'Y', 'U',
        'I', 'O', 'P', '{', '}', '\n', 0, 'A', 'S', 'D', 'F', 'G',
        'H', 'J', 'K', 'L', ':', '"', '~', 0, '|', 'Z', 'X', 'C',
        'V', 'B', 'N', 'M', '<', '>', '?', 0, '*', 0, ' '
    };

    // scancode read from keyboard
    unsigned char scancode;

    // mapped char
    char ch;

    // read scan code from keyboard
    scancode = port_byte_in(0x60);

    //print_hex(scancode);

    // set escaped keys
    if (scancode == 0xE0) {
        keyflag |= E0_SIGN;
        return;
    }

    // unset escaped keys
    if (keyflag & E0_SIGN) {
        keyflag &= ~E0_SIGN;
        return;
    }
    // un shift
    if (scancode & 0x80) {
        keyflag &= ~(shift_code[scancode]);
        return;
    }

    // set shift
    keyflag |= shift_code[scancode];

    // toggle lock
    keyflag ^= lock_code[scancode];

    // get shift keys
    if (keyflag & SHIFT) {
        ch = shift_key_map[scancode];
    }

    // get key
    else {
        ch = key_map[scancode];
    }

    //print_hex(scancode);

    // check for caps
    if (keyflag & CAPS_LOCK) {
        if ('a' <= ch && ch <= 'z')
            ch -= 32;
        else if ('A' <= ch && ch <= 'Z')
            ch += 32;
    }


    // handle backspace
    if (ch == BACKSPACE) {
        if (backspace(key_buffer)) {
            print_backspace();
        }
    }

    // handle enter key
    else if (ch == ENTER) {
        execute_command(key_buffer); // execute command
        key_buffer[0] = '\0'; // clear key buffer

    }

    // any other key store in buffer
    else {

        append(key_buffer, ch);

        // print out letter
        char str[2] = { ch, '\0' };
        print_string(str);
    }

}
```

```
// register keyboard handler at IRQ1
void init_keyboard() {
    register_interrupt_handler(IRQ1, keyboard_callback);
    keyflag=0;
}
```

Command Processor

The kernel now becomes the command processor to process the commands typed in on the keyboard.

The new kernel has to install the ISRs, effectively loading our IDT. Then it will enable external interrupts by setting the interrupt flag using **sti.** Finally, we can call the init_keyboard function that registers the keyboard interrupt handler.

Step 8
Make a new header file called kernel.h and put the following code in it

```
        // kernel.h
        // execute keyboard command
        void execute_command(char* input);
```

Step 9
Make a new file called kernel3.c in Lesson7 folder and put the following code.

```
// kernel3.c
// install the ISRs, loading IDT.
// enable external interrupts by setting the interrupt flag using sti.
// call the init_keyboard function that registers the keyboard interrupt handler.

#include "screen.h"
#include "strings.h"
#include "isr.h"
#include "keyboard.h"

void kernel () {

        //clear_screen();
        //char message[] = "OsX Kernel running in 32 bit protected mode";
        //print_string(message);

        clear_screen();

        char msg1[] = "Installing interrupt service routines (ISRs).\n";
        print_string(msg1);
        isr_install();

        char msg2[] = "Enabling external interrupts.\n";
        print_string(msg2);
        asm volatile("sti");
        char msg3[] = "Initializing keyboard (IRQ 1).\n";
        print_string(msg3);
        init_keyboard();

        char prompt[] = "\nOSX> ";
        print_string(prompt);
}

// receive commands to execute
void execute_command(char* input) {

        // exit command
        char exitmsg[] = "EXIT";
        if (compare_string(input, exitmsg) == 0) {
               char msg[] = "Exit\n";
               print_string(msg);
               asm volatile("hlt"); // halt cpu

        }
        char msg[] = "\ncommand: ";
        print_string(msg);
        print_string(input);
        char prompt[] = "\nOSX> ";
        print_string(prompt);
}
```

Step 10

Increase the bootloader8.asm to load in more 15 or more sectors

```
        LOAD_SECTORS equ 15
```

copy boot2.bat to boot3.bat

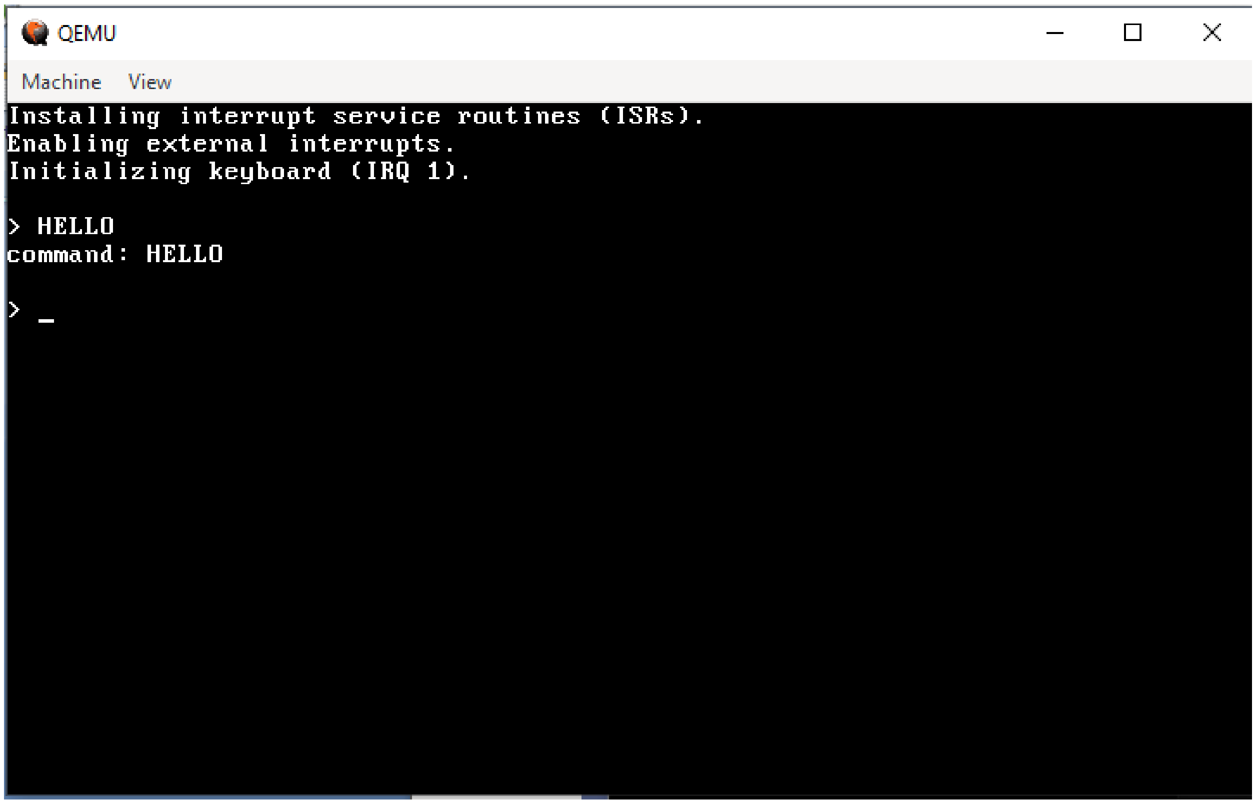
Paste in the following lines rather than updating, so there is no mistakes

Note: Lines `keyboard.o isr.o idt.o interrupts.o are wrapped around`

```
rem boot3.bat
"C:\Program Files\nasm\nasm" -f bin bootloader8.asm -o bootloader.bin
gcc -m32 -ffreestanding -c kernel3.c -o kernel.o
rem objdump -d  kernel.o
gcc -m32 -ffreestanding -c ports.c -o ports.o
gcc -m32 -ffreestanding -c mem.c -o men.o
gcc -m32 -ffreestanding -c screen.c -o screen.o
gcc -m32 -ffreestanding -c strings.c -o strings.o
gcc -m32 -ffreestanding -c keyboard.c -o keyboard.o
gcc -m32 -ffreestanding -c isr.c -o isr.o
gcc -m32 -ffreestanding -c idt.c -o idt.o
"C:\Program Files\nasm\nasm"  -f elf interrupts.asm -o interrupts.o
ld -T NUL -m i386pe -o kernel.tmp -Ttext 0x7e00 kernel.o ports.o mem.o screen.o strings.o
keyboard.o isr.o idt.o interrupts.o
objcopy -O binary -j .text  kernel.tmp kernel.bin
rem "C:\Program Files\nasm\ndisasm" -b 32 kernel.bin
copy /b bootloader.bin+kernel.bin+padding.bin osximage.img
rem "C:\Program Files\nasm\ndisasm" -b 32 osximage.img
"C:\Program Files\qemu\qemu-system-x86_64.exe" osximage.img
```

Run the program by calling boot3.bat in your nasm command prompt.

You should get something like this:



```
Installing interrupt service routines (ISRs).
Enabling external interrupts.
Initializing keyboard (IRQ 1).

> HELLO
command: HELLO

> _
```

Lesson7 homework Question1

In the executeComand function process the following basic shell commands:

Command	Description
CLS	Clear screen
HELP	Print help menu
EXIT	Halt operating System

After the interrupt initialization code call clearScreen and the print out the OsX logo.

In the execute_command function make the help menu and process the above commands.

Use the compare_string to compare commands..

You need to make string variables when comparing commands rather than using string literals like `compare_string(input, "HELP")`

```
           char helpmsg[] = "HELP";
           compare_string(input, exitmsg)
```

```
***EACH MENU ITEM MUST HAVE ITS OWN MESSSAGE***
***EACH STRING CAN OMLY HANFEL 31 CHARACTERS***
```

Make a **tolower_string** and a **toupper_string** functions so you can compare upper and lower case.

```
            char* toupper_string(char s[]);
            char* tolower_string(char s[]);
```

Put the **tolower_string** AND a **toupper_string** functions in files strings.h and strings.c from Lesson 6.

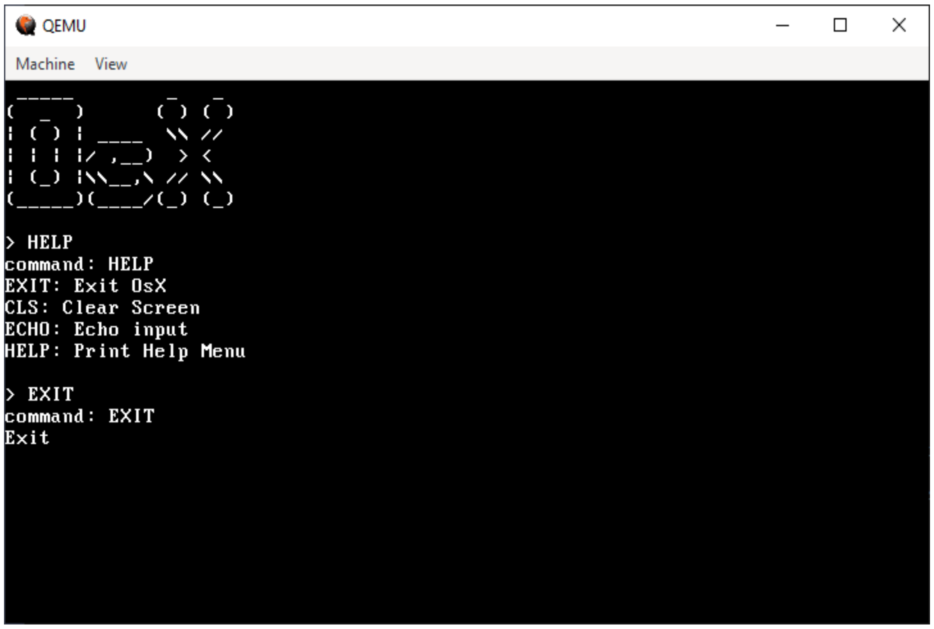
Add value 32 to change upper case to lower case. Subtract value 32 to change upper case to lower case.

Use the toupper_string function to change the input command line to upper case

```
             toupper_string(input)
```

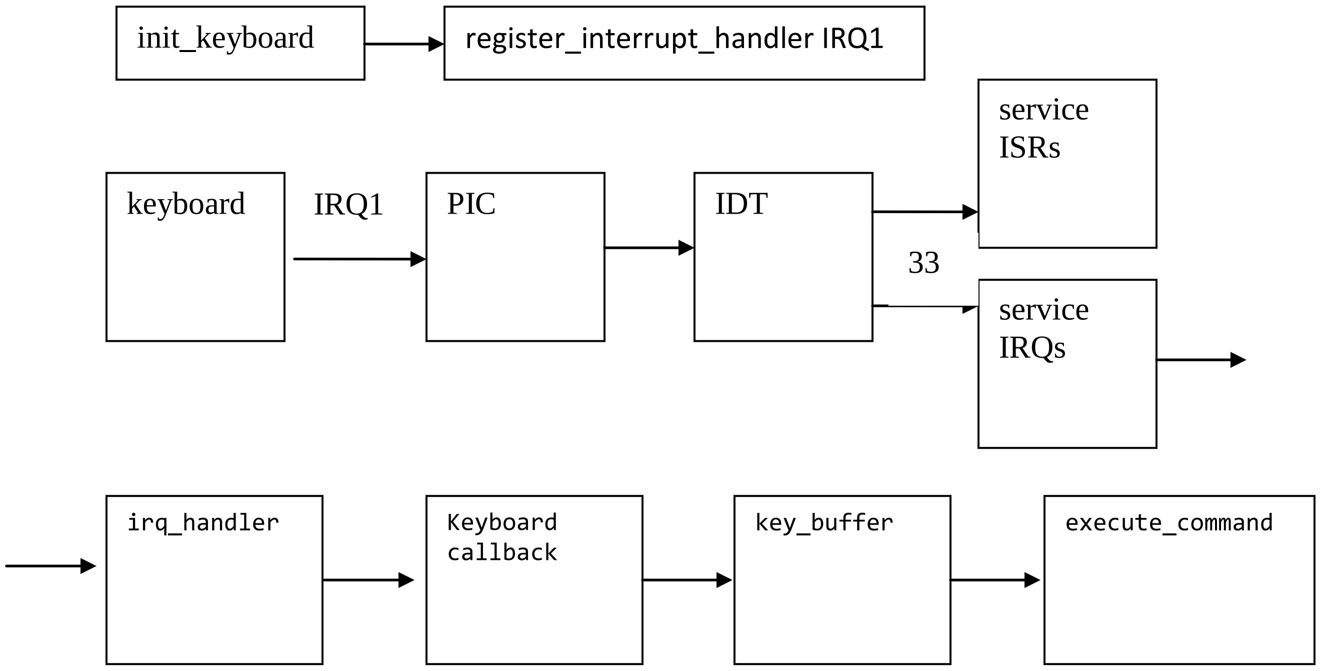
Put everything in your kernel3.c file.

You should get something like this:



```
 _____        _   _
(  _  )      ( ) ( )
| ( ) | ____  \\ //
| | | |/ ,__)  > <
| (_) |\\__,\ // \\
(_____)(____/(_) (_)

> HELP
command: HELP
EXIT: Exit OsX
CLS: Clear Screen
ECHO: Echo input
HELP: Print Help Menu

> EXIT
command: EXIT
Exit
```

The keyboard mechanism works like this:



Adding a Timer interrupt

The 8253 **programmable interval timer** (**PIT**) is a counter that generates an output signal when it reaches a programmed count.

The 8253 PITs) has three 16-bit counters.[1]

Timer Channel 0 is assigned to IRQ-0 the highest priority hardware interrupt. Timer Channel 1 is assigned to DRAM refresh and Timer Channel 2 is assigned to the PC speaker.

Channel 0

The output from PIT channel 0 is connected to the PIC chip, so that it generates an "IRQ 0". Typically during boot the BIOS sets channel 0 with a count of 65535 or 0 (which translates to 65536), which gives an output frequency of 18.2065 Hz (or an IRQ every 54.9254 ms). Channel 0 is probably the most useful PIT channel and is used to generate an infinite series of "timer ticks" at a frequency of your choice.

The frequency must be higher than 18 Hz. Channel 0 is also used to generate single CPU interrupts in "one shot" mode after programmable short delays that are less than an 18th of a second.

The PIT chip uses the following I/O ports:

```
I/O port      Usage
0x40          Channel 0 data port (read/write)
0x41          Channel 1 data port (read/write)
0x42          Channel 2 data port (read/write)
0x43          Mode/Command register (write only, a read is ignored)
```

The Mode/Command register at I/O address 0x43 contains the following bits:

```
Bits          Usage
6 and 7       Select channel :
                 0 0 = Channel 0
                 0 1 = Channel 1
                 1 0 = Channel 2
                 1 1 = Read-back command (8254 only)
4 and 5       Access mode :
                 0 0 = Latch count value command
                 0 1 = Access mode: lobyte only
                 1 0 = Access mode: hibyte only
                 1 1 = Access mode: lobyte/hibyte
```

```
1 to 3       Operating mode :
                0 0 0 = Mode 0 (interrupt on terminal count)
                0 0 1 = Mode 1 (hardware re-triggerable one-shot)
                0 1 0 = Mode 2 (rate generator)
                0 1 1 = Mode 3 (square wave generator)
                1 0 0 = Mode 4 (software triggered strobe)
                1 0 1 = Mode 5 (hardware triggered strobe)
                1 1 0 = Mode 2 (rate generator, same as 010b)
                1 1 1 = Mode 3 (square wave generator, same as 011b)
0            BCD/Binary mode: 0 = 16-bit binary, 1 = four-digit BCD
```

The timer we will be handled by our IRQ0 service routine using number 32.

The PIT counter 0 is set to 1193180 counts per second, If we divide by 11931 we can get 100 counts per second.

Evert time the timer times out a tick variable will be incremented.

Step 11

Make a new header file called timer.h and put the following code in it

```
// timer.h
#pragma once

// timer functions
void init_timer(unsigned int freq);
```

Step 12

Make a companion implementation C file called timer.c and put the following code in it

```
// timer.c

// count timer ticks

#include "timer.h"
#include "ports.h"
#include "isr.h"
#include "screen.h"
```

```
unsigned int tick = 0;
// increment timer tick
static void timer_callback(registers_t* regs) {

    tick++;

    //char tickMsg[] = "Tick: ";
    //print_string(tickMsg);
}

// initialize timer
void init_timer(unsigned int freq) {

    // register timer interrupt to IRQ0 service routine
    register_interrupt_handler(IRQ0, timer_callback);

    // set up the desired interval using PIT hardware clock at 1193180 Hz */
    unsigned int divisor = 1193180 / freq;
    unsigned char low = (unsigned char)(divisor & 0xFF);
    unsigned char high = (unsigned char)((divisor >> 8) & 0xFF);

    // program PIT
    port_byte_out(0x43, 0x36); // Command port
    port_byte_out(0x40, low);
    port_byte_out(0x40, high);
}
```

Step 13

In kernel3.c file right after the keyboard initialization call the init_timer function with 11931, t his will give you 100 counts per second.

```
init_timer(11931);
```

Put #include time.h at the top of kernel.c

Lesson7 Homework Question2

Add 2 functions to access the timer in file timer.h and timer .c

```
    unsigned int getTick()

and

    void setTick(unsigned int t)
```

In your execute_command function, make a command called TICK that will print out the value of the tick obtained from the timer.

Also make sure you add the TICK command to your help menu.

```
        char menu5[] = "TICK: print tick\n";
        print_string(menu5);
```

And then printout the timer tick value once the TICK command is received using the `getTick()` function

```
        char tickMsg[] = "Tick: ";
        print_string(tickMsg);
        unsigned int tick = getTick();
        print_int(tick);
        print_nl();
```

You will also need to update the `irq_handler` function in file isr.c to the following condition (`r->int_no > 32` to `r->int_no >= 32)` to catch timer interrupt 32.

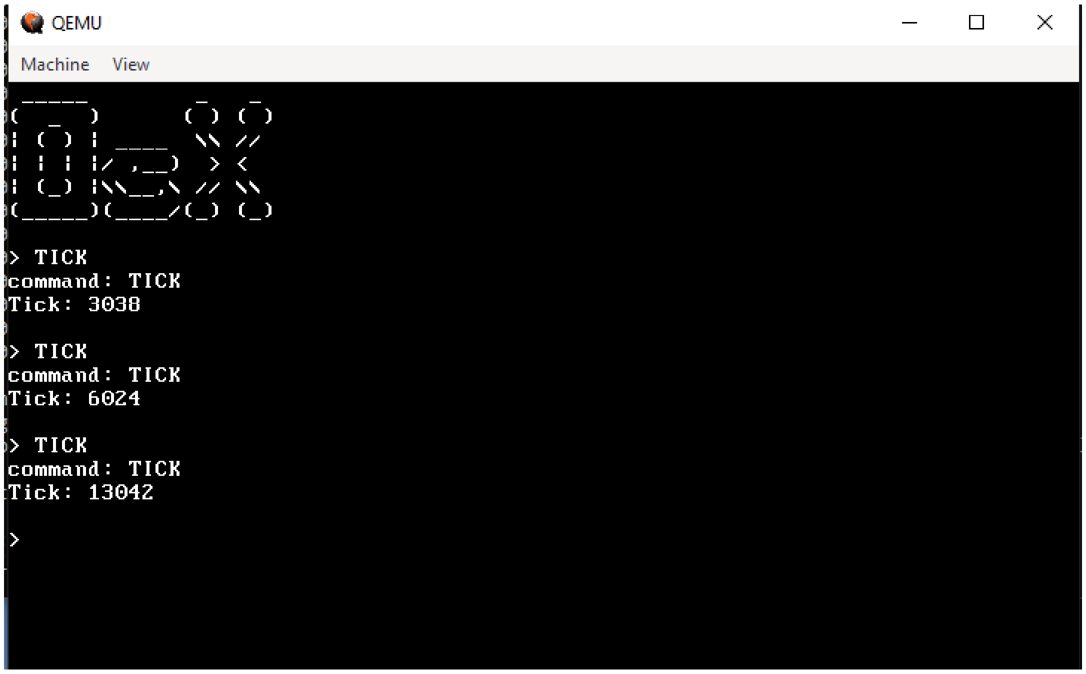
```
    // Handle the interrupt
    if (interrupt_handlers[r->int_no] !=0 && r->int_no >= 32) {

        isr_t handler = interrupt_handlers[r->int_no];
        handler(r);
    }
```

Before running your program you will also need to update boot3 to compile and link the timer.c module.

Note: Lines `keyboard.o isr.o idt.o interrupts.o and timer.o are wrapped around`

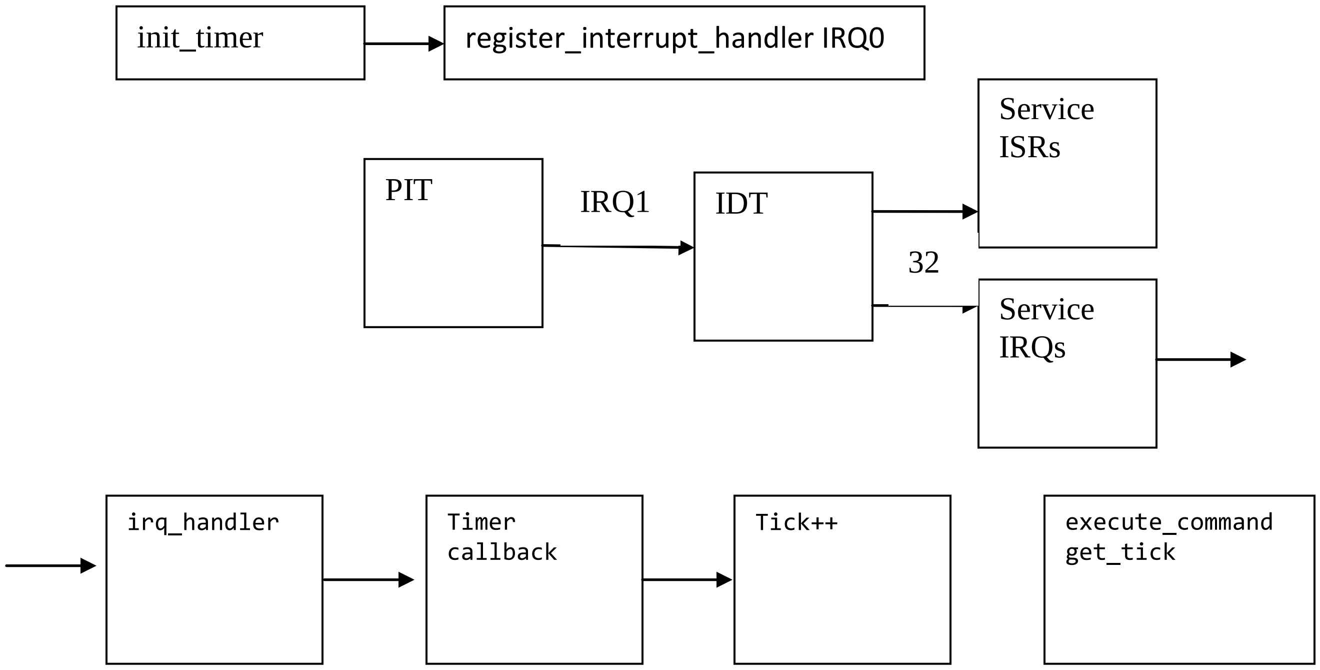
```
rem boot3.bat
"C:\Program Files\nasm\nasm" -f bin bootloader8.asm -o bootloader.bin
gcc -m32 -ffreestanding -c kernel3.c -o kernel.o
rem objdump -d  kernel.o
gcc -m32 -ffreestanding -c ports.c -o ports.o
gcc -m32 -ffreestanding -c mem.c -o men.o
gcc -m32 -ffreestanding -c screen.c -o screen.o
gcc -m32 -ffreestanding -c strings.c -o strings.o
gcc -m32 -ffreestanding -c keyboard.c -o keyboard.o
gcc -m32 -ffreestanding -c isr.c -o isr.o
gcc -m32 -ffreestanding -c idt.c -o idt.o
"C:\Program Files\nasm\nasm" -f elf interrupts.asm -o interrupts.o
gcc -m32 -ffreestanding -c timer.c -o timer.o
ld -T NUL -m i386pe -o kernel.tmp -Ttext 0x7e00 kernel.o ports.o mem.o screen.o strings.o
keyboard.o isr.o idt.o interrupts.o timer.o
objcopy -O binary -j .text  kernel.tmp kernel.bin
rem "C:\Program Files\nasm\ndisasm" -b 32 kernel.bin
copy /b bootloader.bin+kernel.bin+padding.bin osximage.img
rem "C:\Program Files\nasm\ndisasm" -b 32 osximage.img
"C:\Program Files\qemu\qemu-system-x86_64.exe" osximage.img
```

Run the program and you should get something like this:



```
 _____        _   _
(  _  )      ( ) ( )
| ( ) | ____  \\ //
| | | |/ ,__)  > <
| (_) |\\__,\ // \\
(_____)(____/(_) (_)

> TICK
command: TICK
Tick: 3038

> TICK
command: TICK
Tick: 6024

> TICK
command: TICK
Tick: 13042

>
```

The Timer mechanism works like this:



Playing Music

The PC Speaker can be connected directly to the output of timer number 2 on the Programmable Interval Timer by setting bit 0 of port 0x61 (=1). In this mode, when the timer goes "high" (=1) the speaker will move to the "out" position. Likewise, when the timer goes "low" (=0) the speaker will move to the "in" position. By changing the frequency at which timer 2 "ticks", the PC Speaker can be made to output sound of the same frequency.

Here are the steps to play music

Step 1

Make a headers file called sounds.h and put the following function prototype in it.

```
// sounds.h
void play_sound();
```

Step 2

Make a code file called sounds.c and put the following codes in it.

```
// sounds.c
#include "sounds.h"
#include "screen.h"
#include "ports.h"
#include "timer.h"

 // Play sound using built-in speaker
 // qemu sound setup
 // -audiodev sdl,id=speaker -machine pcspk-audiodev=speaker

// private sound functions
void play_freq(unsigned int nFrequence);
void nosound();
void time_wait(int duration );
void beep(unsigned int freq, int duration); {
```

```
// play frequency
void play_freq(unsigned int nFrequence) {
    unsigned int Div;
    unsigned char tmp;

    //Set the PIT to the desired frequency
    Div = 1193180 / nFrequence;
    port_byte_out(0x43, 0xb6);
    port_byte_out(0x42, (unsigned char) (Div) );
    port_byte_out(0x42, (unsigned char) (Div >> 8));

    // play the sound using the PC speaker
    tmp = port_byte_in(0x61);
    if (tmp != (tmp | 3)) {
        port_byte_out(0x61, tmp | 3);
    }
 }

 // turn speaker off
void nosound() {
    unsigned char tmp = port_byte_in(0x61) & 0xFC;
    port_byte_out(0x61, tmp);
 }

// duration delay
void time_wait(int duration )
{
   for(int i=0;i<duration;i++)
   {
    for(int j=0;j<100000;j++)
    {

    }
   }
}
```

```
 //Make a beep for frequency and duration
 void beep(unsigned int freq, int duration) {
     play_freq(freq);
     time_wait(duration);
     nosound();
 }

// play sound
void play_sound()
{
char msg[] = "play sound";

print_string(msg);

// test beep
beep(1000,1000);

    // Loop for sound Jingle
    for (int x = 0; x < 2; x++) {
        beep(523, 500);
    }

    // sound Bell
    beep(523, 800);
    time_wait(200);

    // Loop for sound Jingle
    for (int x = 0; x < 2; x++) {
        beep(523, 500);
    }

    // sound Bell
    beep(523, 800);

    // Sound for rest of the tone
    time_wait(200);
    beep(523, 500);
    time_wait(50);
```

```
beep(659, 400);
time_wait(50);
beep(440, 400);
time_wait(50);
beep(494, 400);
time_wait(50);
beep(523, 750);
time_wait(400);
beep(600, 400);
time_wait(100);
beep(600, 350);
time_wait(200);
beep(600, 300);
time_wait(150);
beep(600, 250);
time_wait(150);
beep(600, 150);
time_wait(150);
beep(550, 250);
time_wait(150);
beep(555, 350);
time_wait(50);
beep(555, 200);
time_wait(150);
beep(555, 200);
time_wait(150);
beep(690, 200);
time_wait(150);
beep(690, 200);
time_wait(150);
beep(610, 200);
time_wait(150);
beep(535, 160);
time_wait(100);
beep(500, 150);
beep(500, 50);
time_wait(200);
beep(700, 200);
```

```
char msg2[] = "done";
print_string(msg2);

// restore timer
init_timer(10000);
}
```

sound functions	description
`void play_freq (unsigned int nFrequence)`	Plays note at a certain frequency
`void nosound()`	Turns speaker off
`void time_wait(int duration )`	Time delay
`void beep (unsigned int freq, int duration)`	Plat node for a certain frequency and duration

Step 3

In kernel3.c put #include “sounds.h” at the top and add a sound menu and call the playsound function

```
     char msg6[] = "SOUND";

    char menu6[] = "SOUND: play sound\n";
        print_string(menu17);


    else if (compare_string(input, msg17) == 0)
        {
        play_sound();
        }
```

Step 4

Make a new bat file called music.bat as follows:

```
; rem music.bat
"C:\Program Files\nasm\nasm" -f bin bootloader8.asm -o bootloader.bin
gcc -m32 -ffreestanding -c kernel3.c -o kernel.o
rem objdump -d  kernel.o
gcc -m32 -ffreestanding -c ports.c -o ports.o
gcc -m32 -ffreestanding -c mem.c -o mem.o
gcc -m32 -ffreestanding -c screen.c -o screen.o
gcc -m32 -ffreestanding -c strings.c -o strings.o
gcc -m32 -ffreestanding -c keyboard.c -o keyboard.o
gcc -m32 -ffreestanding -c isr.c -o isr.o
gcc -m32 -ffreestanding -c idt.c -o idt.o
"C:\Program Files\nasm\nasm" -f elf interrupts.asm -o interrupts.o
gcc -m32 -ffreestanding -c timer.c -o timer.o
gcc -m32 -ffreestanding -c sounds.c -o sounds.o
ld -T NUL -m i386pe -o kernel.tmp -Ttext 0x7e00 kernel.o ports.o mem.o
screen.o strings.o keyboard.o isr.o idt.o interrupts.o timer.o
sounds.o
objcopy -O binary -j .text  kernel.tmp kernel.bin
rem ndisasm -b 32 kernel.bin
copy /b bootloader.bin+kernel.bin+padding.bin osximage.img
rem ndisasm -b 32 osximage.img
"C:\Program Files\qemu\qemu-system-x86_64.exe" -audiodev
sdl,id=speaker -machine pcspk-audiodev=speaker -drive
file=osximage.img,format=raw
```

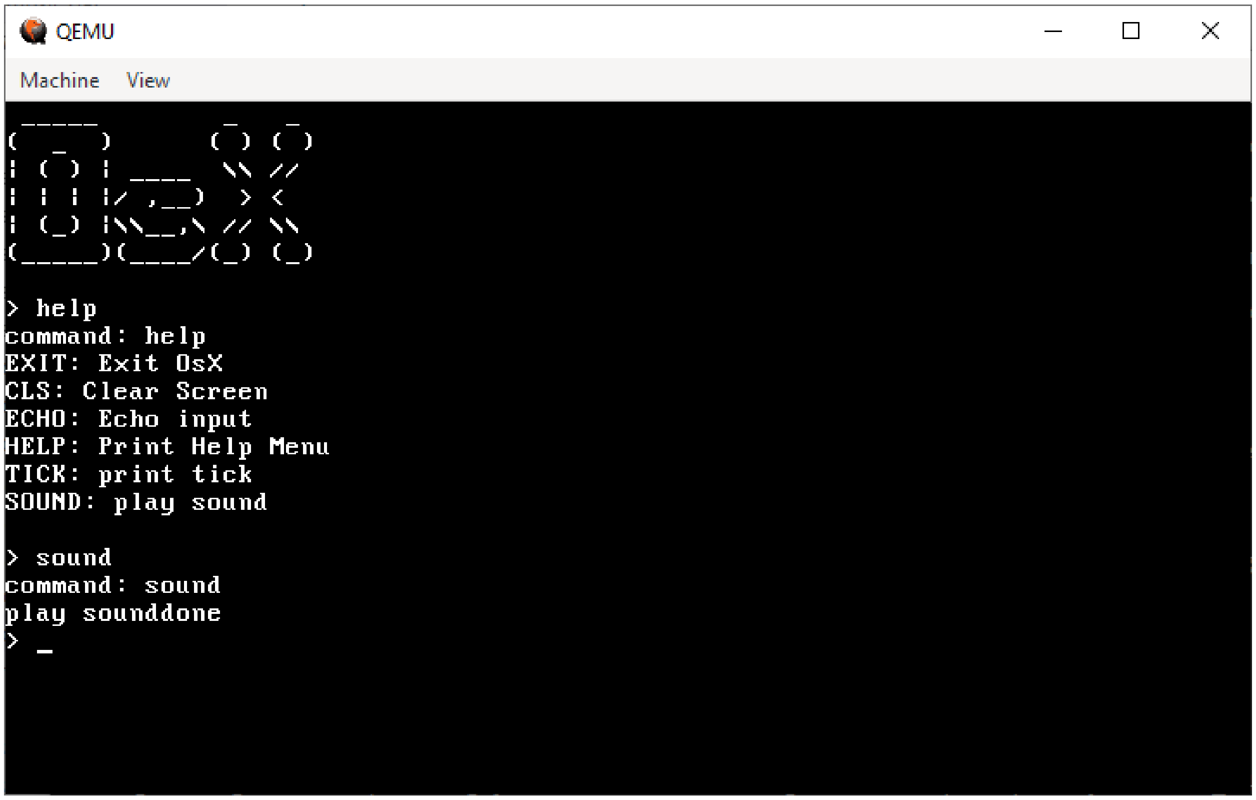
The music song should play on the pc speaker or headphones on newer pc

The line sets up the audio device sdl:

```
audiodev sdl,id=speaker -machine pcspk-audiodev=speaker
directs the notes to the speaker
```

Step 5

Run the music.bat file and select sound and you will get something like this:



```
 _____        _   _
(  _  )      ( ) ( )
| ( ) | ____  \\ //
| | | |/ ,__)  > <
| (_) |\\__,\ // \\
(_____)(____/(_) (_)

> help
command: help
EXIT: Exit OsX
CLS: Clear Screen
ECHO: Echo input
HELP: Print Help Menu
TICK: print tick
SOUND: play sound

> sound
command: sound
play sounddone
> _
```

todo:

Make your own song using beep function.

End Lesson 7