Worksheet 2 ‑ Inputs & Interrupts

Worksheet 2 is split into two parts, this is the second part. Each part is worth 50% of the total for worksheet 2, which itself is worth 50% of the module assessment, with worksheet 1 making up the other 50%.

Marking scheme

* Task 1 ‑ 20%
* Task 2 ‑ 20%
* Task 3 ‑ 40%
* Readme ‑ 20%

All work should be documented in the worksheets README.md.

# Introduction

In the previous worksheet, we implemented a framebuffer to write characters to the terminal. This worksheet builds on that foundation by implementing keyboard input handling through interrupts. We’ll be following Chapter 6 of the OS handbook, which covers interrupt handling and basic I/O oper‑ ations.

# Core Concepts

## Interrupts Overview

Interrupts are signals to the processor indicating that it should pause its current execution to han‑ dle some immediate task (like keyboard input). In this worksheet, we’ll focus on handling keyboard interrupts through the Programmable Interrupt Controller (PIC).

## Input/Output (I/O)

Our OS needs to communicate with hardware devices. We’ll use I/O ports to read keyboard input and configure the interrupt controller. This requires both assembly‑level port access and higher‑level C code to process the data.

# Implementation Steps

## Extending the I/O Library

First, we’ll add the ability to read input from I/O ports. Add this function to your **io.s** file:

global inb

*; inb - returns a byte from the given I/O port*

*; stack: [esp + 4] The address of the I/O port*

*;*

inb:

*[esp*

*] The return address*

mov **dx, [esp +** 4**]**

↪ *register*

in **al, dx**

↪ *in the al register*

**ret**

*; move the address of the I/O port to the dx*

*; read a byte from the I/O port and store it*

*; return the read byte*

Update your **io.h** header with:

unsigned char inb**(**unsigned short port**);**

## Setting Up Type Definitions

Create a new file **types.h** for consistent data types:

*#ifndef INCLUDE\_TYPES\_H #define INCLUDE\_TYPES\_H*

**typedef** unsigned int u32int**; typedef** int s32int**; typedef** unsigned short u16int**; typedef** short s16int**; typedef** unsigned char u8int**; typedef** char s8int**;**

*/\* Frame buffer colors \*/ #define BLACK* *0*

*#define BLUE* *1*

*#define LIGHT\_GREY 7*

*#endif*

## Programmable Interrupt Controller (PIC) Setup

The PIC handles external interrupts from devices. Create **pic.h** and **pic.c** with the provided code. Key functions include: ‑ pic\_remap(): Reconfigures interrupt numbers to avoid conflicts ‑ pic\_acknowledge(): Signals completion of interrupt handling

*#ifndef INCLUDE\_PIC\_H #define INCLUDE\_PIC\_H*

*#include* "drivers/type.h"

*/\**

*#define PIC\_1 #define PIC\_2*

*I/O port \*/*

*0x20*

*0xA0*

*/\* IO base address for master PIC \*/*

*/\* IO base address for slave PIC \*/*

*#define PIC\_1\_COMMAND PIC\_1 #define PIC\_1\_DATA (PIC\_1+1) #define PIC\_2\_COMMAND PIC\_2 #define PIC\_2\_DATA (PIC\_2+1)*

*#define PIC\_1\_OFFSET 0x20 #define PIC\_2\_OFFSET 0x28*

*#define PIC\_2\_END PIC\_2\_OFFSET + 7*

*#define PIC\_1\_COMMAND\_PORT 0x20 #define PIC\_2\_COMMAND\_PORT 0xA0*

*#define PIC\_ACKNOWLEDGE 0x20*

*#define PIC\_ICW1\_ICW4 #define PIC\_ICW1\_SINGLE #define PIC\_ICW1\_INTERVAL4*

*#define PIC\_ICW1\_LEVEL*

*0x01 /\* ICW4 (not) needed \*/ 0x02 /\* Single (cascade) mode \*/*

*0x04 /\* Call address interval 4 (8) \*/*

*0x08 /\* Level triggered (edge) mode \*/*

void pic\_remap**(**s32int offset1**,** s32int offset2**);**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#define* | *PIC\_ICW1\_INIT* | *0x10* | */\** | *Initialization - required! \*/* |
| *#define* | *PIC\_ICW4\_8086* | *0x01* | */\** | *8086/88 (MCS-80/85) mode \*/* |
| *#define* | *PIC\_ICW4\_AUTO* | *0x02* | */\** | *Auto (normal) EOI \*/* |
| *#define* | *PIC\_ICW4\_BUF\_SLAVE* | *0x08* | */\** | *Buffered mode/slave \*/* |
| *#define* | *PIC\_ICW4\_BUF\_MASTER* | *0x0C* | */\** | *Buffered mode/master \*/* |
| *#define* | *PIC\_ICW4\_SFNM* | *0x10* | */\** | *Special fully nested (not) \*/* |

void pic\_acknowledge**(**u32int interrupt**);**

*#endif /\* INCLUDE\_PIC\_H \*/*

## Interrupt Descriptor Table (IDT)

Create the interrupt handling infrastructure with these files: ‑ **interrupts.h**: Defines the structures for interrupt descriptors ‑ **interrupt\_handlers.s**: Contains assembly‑level interrupt handling ‑ **inter‑ rupt\_asm.s**: Defines individual interrupt handlers ‑ **interrupts.c**: Implements the C‑level interrupt handling

Interrupts are handled by the IDT (Interrupts Descriptor Table), as suggested in the OS handbook we are going to use structs to define each of these entries in the IDT. This should be added to a new file called **interrupts.h**.

*#ifndef INCLUDE\_INTERRUPTS #define INCLUDE\_INTERRUPTS*

*#include* "type.h"

**struct** IDT

**{**

u16int size**;** u32int address**;**

**}**  attribute **((**packed**));**

**struct** IDTDescriptor **{**

*/\* The lowest 32 bits \*/*

u16int offset\_low**;** *// offset bits 0..15*

u16int segment\_selector**;** *// a code segment selector in GDT or LDT*

*/\* The highest 32 bits \*/*

u8int reserved**;** *// Just 0.*

u8int type\_and\_attr**;** *// type and attributes*

u16int offset\_high**;** *// offset bits 16..31*

**}**  attribute **((**packed**));**

**struct** cpu\_state **{** u32int eax**;** u32int ebx**;**

u32int ecx**;** u32int edx**;** u32int ebp**;** u32int esi**;** u32int edi**;**

**}**  attribute **((**packed**));**

**struct** stack\_state **{** u32int error\_code**;** u32int eip**;**

u32int cs**;** u32int eflags**;**

**}**  attribute **((**packed**));**

void interrupt\_handler**(struct** cpu\_state cpu**,** u32int interrupt**, struct**

↪ stack\_state stack**);**

void interrupts\_install\_idt**();**

*// Wrappers around ASM.*

void load\_idt**(**u32int idt\_address**);** void interrupt\_handler\_33**();**

void interrupt\_handler\_14**();**

*#endif /\* INCLUDE\_INTERRUPTS \*/*

Now we have created ptototypes for three functions in the **interrupts.h** file. These are functions writ‑ ten in assembler that we also need to define.

void load\_idt**(**u32int idt\_address**);** void interrupt\_handler\_33**();**

void interrupt\_handler\_14**();**

For the function **load\_idt** lets create **interrupt\_handlers.s**

global load\_idt

*; load\_idt - Loads the interrupt descriptor table (IDT).*

*; stack: [esp + 4] the address of the first entry in the IDT*

*; [esp ] the return address*

load\_idt:

mov **eax, [esp +** 4**]** lidt **[eax]**

**ret**

Now for our interrupt handler routines it’s slightly more complicated as we need to define a handler for every interrupt even though we are only really interested in the keyboard input right now. To do this we can create the assembler file **interrupt\_asm.s**. This will define a number of interrupts with the name **interrupt\_handler\_x**, however we only really need the prototype **void interrupt\_handler\_33();** as this is for the keyboard.

*;Generic Interrupt Handler*

*;*

extern interrupt\_handler

%*macro* no\_error\_code\_interrupt\_handler 1 global interrupt\_handler\_**%**1 interrupt\_handler\_**%**1**:**

push dword 0

push dword **%**1

**jmp** common\_interrupt\_handler

%endmacro

*; push 0 as error code*

*; push the interrupt number*

*; jump to the common handler*

%*macro* error\_code\_interrupt\_handler 1 global interrupt\_handler\_**%**1

interrupt\_handler\_**%**1**:**

push

**jmp**

%endmacro

dword **%**1

common\_interrupt\_handler

*; push the interrupt number*

*; jump to the common handler*

common\_interrupt\_handler:

↪ *interrupt handler*

*; save the registers*

*; the common parts of the generic*

push **eax**

push **ebx**

push **ecx**

push **edx**

push **ebp**

push **esi**

push **edi**

*; call the C function*

**call** interrupt\_handler

*; restore the registers*

pop **edi** pop **esi** pop **ebp** pop **edx** pop **ecx** pop **ebx**

pop **eax**

*; restore the esp*

add **esp,** 8

*; return to the code that got interrupted*

**iret**

no\_error\_code\_interrupt\_handler 33 *; create handler for interrupt 1*

↪ *(keyboard)*

Next we are going to need some code to handle enabling and disabling of interrupts in our OS. Lets ass the following files to our repo. You could equally call **sti** and **cli** directly but this makes the code more readable.

## hardware\_interrupt\_enabler.h

*#ifndef INCLUDE\_HARDWARE\_INTERRUPT\_ENABLER\_H #define INCLUDE\_HARDWARE\_INTERRUPT\_ENABLER\_H*

void enable\_hardware\_interrupts**();** void disable\_hardware\_interrupts**();**

*#endif /\* INCLUDE\_USER\_MODE\_H \*/*

**hardware\_interrupt\_enabler.s**

global enable\_hardware\_interrupts

enable\_hardware\_interrupts:

sti

**ret**

global disable\_hardware\_interrupts disable\_hardware\_interrupts:

cli

**ret**

Now finally we can create our **interrupts.c** which contains the code to setup the interrutps themself and provide a handler to get data from the keyboard itself.

## interrupts.c

*#include* "interrupts.h" *#include* "pic.h" *#include* "io.h"

*#include* "frame\_buffer.h"

*#include* "keyboard.h"

*#define INTERRUPTS\_DESCRIPTOR\_COUNT 256*

*#define INTERRUPTS\_KEYBOARD 33*

*#define INPUT\_BUFFER\_SIZE 256*

u8int input\_buffer**[**INPUT\_BUFFER\_SIZE**];** u8int buffer\_index **=** 0**;**

**struct** IDTDescriptor idt\_descriptors**[**INTERRUPTS\_DESCRIPTOR\_COUNT**]; struct** IDT idt**;**

u32int BUFFER\_COUNT**;**

void interrupts\_init\_descriptor**(**s32int index**,** u32int address**)**

**{**

idt\_descriptors**[**index**].**offset\_high **= (**address **>>** 16**) &** 0xFFFF**;** *// offset*

↪ *bits 0..15*

idt\_descriptors**[**index**].**offset\_low **= (**address **&** 0xFFFF**);** *// offset bits*

↪ *16..31*

idt\_descriptors**[**index**].**segment\_selector **=** 0x08**;** *// The second (code)*

↪ *segment selector in GDT: one segment is 64b.*

idt\_descriptors**[**index**].**reserved **=** 0x00**;** *// Reserved.*

*/\**

*Bit: | 31 16 | 15 | 14 13 | 12 | 11 10 9 8 | 7 6*

↪ *5 | 4 3 2 1 0 |*

*Content: | offset high | P | DPL | S | D and GateType | 0*

↪ *0 0 | reserved*

*P If the handler is present in memory or not (1 = present, 0 = not*

↪ *present). Set to 0 for unused interrupts or for Paging.*

*DPL Descriptor Privilige Level, the privilege level the handler can*

↪ *be called from (0, 1, 2, 3).*

*S Storage Segment. Set to 0 for interrupt gates. D Size of gate, (1 = 32 bits, 0 = 16 bits).*

*\*/*

idt\_descriptors**[**index**].**type\_and\_attr **= (**0x01 **<<** 7**) |** *// P*

**(**0x00 **<<** 6**) | (**0x00 **<<** 5**) |** *// DPL*

0xe**;** *// 0b1110=0xE 32-bit interrupt gate*

**}**

void interrupts\_install\_idt**()**

**{**

interrupts\_init\_descriptor**(**INTERRUPTS\_KEYBOARD**, (**u32int**)**

↪ interrupt\_handler\_33**);**

idt**.**address **= (**s32int**) &**idt\_descriptors**;**

idt**.**size **= sizeof(struct** IDTDescriptor**) \*** INTERRUPTS\_DESCRIPTOR\_COUNT**;** load\_idt**((**s32int**) &**idt**);**

*/\*pic\_remap(PIC\_PIC1\_OFFSET, PIC\_PIC2\_OFFSET);\*/*

pic\_remap**(**PIC\_1\_OFFSET**,** PIC\_2\_OFFSET**);**

*// Unmask keyboard interrupt (IRQ1)*

outb**(**0x21**,** inb**(**0x21**) & ~(**1 **<<** 1**));**

**}**

*/\* Interrupt handlers*

↪ *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

void interrupt\_handler**(** attribute **((**unused**)) struct** cpu\_state cpu**,** u32int

↪ interrupt**,**  attribute **((**unused**)) struct** stack\_state stack**) {**

u8int input**;**

u8int ascii**;**

**switch (**interrupt**) {**

**case** INTERRUPTS\_KEYBOARD**:**

**while ((**inb**(**0x64**) &** 1**)) {**

input **=** keyboard\_read\_scan\_code**();**

*// Only process if it's not a break code*

**if (!(**input **&** 0x80**)) {**

**if (**input **<=** KEYBOARD\_MAX\_ASCII**) {**

ascii **=** keyboard\_scan\_code\_to\_ascii**(**input**); if (**ascii **!=** 0**) {**

*// We have detected a backspace*

**if (**ascii **==** '\b'**) {**

*// Remove the last character*

**}**

*// We have detected a newline*

**else if (**ascii **==** '\n'**) {**

*// Move our position to a newline*

**}**

*// We have detected a regular character*

**else {**

*// Add the new character to the display*

**}**

**}**

**}**

**}**

buffer\_index **= (**buffer\_index **+** 1**) %** INPUT\_BUFFER\_SIZE**;**

**}**

pic\_acknowledge**(**interrupt**); break;**

**default:**

**break;**

**}**

**}**

## Keyboard Input Processing

Implement keyboard handling in **keyboard.h** and **keyboard.c**. These files manage: ‑ Reading scan codes from the keyboard ‑ Converting scan codes to ASCII characters ‑ Handling special keys (backspace, enter)

## keyboard.h

*#ifndef INCLUDE\_KEYBOARD\_H #define INCLUDE\_KEYBOARD\_H*

*#define KEYBOARD\_MAX\_ASCII 83*

*#include* "drivers/type.h"

u8int keyboard\_read\_scan\_code**(**void**);** u8int keyboard\_scan\_code\_to\_ascii**(**u8int**);** *#endif /\* INCLUDE\_KEYBOARD\_H \*/*

**keyboard.c**

*#include* "io.h"

*#include* "frame\_buffer.h"

*#define KEYBOARD\_DATA\_PORT 0x60*

*/\*\* read\_scan\_code:*

* *Reads a scan code from the keyboard*

*\**

* ***@return*** *The scan code (NOT an ASCII character!)*

*\*/*

u8int keyboard\_read\_scan\_code**(**void**)**

**{**

**return** inb**(**KEYBOARD\_DATA\_PORT**);**

**}**

u8int keyboard\_scan\_code\_to\_ascii**(**u8int scan\_code**)**

**{**

*// Ignore key releases (scan codes with bit 7 set)*

**if (**scan\_code **&** 0x80**) {**

**return** 0**;**

**}**

*// Scan code to ASCII mapping for standard US QWERTY keyboard*

**switch(**scan\_code**) {**

*// Numbers row*

**case** 0x02**: return** '1'**; case** 0x03**: return** '2'**; case** 0x04**: return** '3'**; case** 0x05**: return** '4'**; case** 0x06**: return** '5'**; case** 0x07**: return** '6'**; case** 0x08**: return** '7'**; case** 0x09**: return** '8'**; case** 0x0A**: return** '9'**; case** 0x0B**: return** '0'**; case** 0x0C**: return** '-'**; case** 0x0D**: return** '='**;**

**case** 0x0E**: return** '\b'**;** *// Backspace*

*// Top letter row*

**case** 0x10**: return** 'q'**; case** 0x11**: return** 'w'**; case** 0x12**: return** 'e'**; case** 0x13**: return** 'r'**; case** 0x14**: return** 't'**; case** 0x15**: return** 'y'**; case** 0x16**: return** 'u'**; case** 0x17**: return** 'i'**; case** 0x18**: return** 'o'**; case** 0x19**: return** 'p'**; case** 0x1A**: return** '['**; case** 0x1B**: return** ']'**;**

**case** 0x1C**: return** '\n'**;** *// Enter*

*// Middle letter row* **case** 0x1E**: return** 'a'**; case** 0x1F**: return** 's'**; case** 0x20**: return** 'd'**; case** 0x21**: return** 'f'**; case** 0x22**: return** 'g'**;**

**case** 0x23**: return** 'h'**; case** 0x24**: return** 'j'**; case** 0x25**: return** 'k'**; case** 0x26**: return** 'l'**; case** 0x27**: return** ';'**; case** 0x28**: return** '\''**; case** 0x29**: return** '`'**;**

*// Bottom letter row* **case** 0x2B**: return** '\\'**; case** 0x2C**: return** 'z'**; case** 0x2D**: return** 'x'**; case** 0x2E**: return** 'c'**; case** 0x2F**: return** 'v'**; case** 0x30**: return** 'b'**; case** 0x31**: return** 'n'**; case** 0x32**: return** 'm'**; case** 0x33**: return** ','**; case** 0x34**: return** '.'**; case** 0x35**: return** '/'**;**

**case** 0x39**: return** ' '**;** *// Space bar*

*// Numpad*

**case** 0x37**: return** '\*'**;** *// Numpad \**

**case** 0x47**: return** '7'**; case** 0x48**: return** '8'**; case** 0x49**: return** '9'**; case** 0x4A**: return** '-'**; case** 0x4B**: return** '4'**; case** 0x4C**: return** '5'**; case** 0x4D**: return** '6'**; case** 0x4E**: return** '+'**; case** 0x4F**: return** '1'**; case** 0x50**: return** '2'**; case** 0x51**: return** '3'**; case** 0x52**: return** '0'**; case** 0x53**: return** '.'**;**

**default: return** 0**;** *// Unknown scan code*

**}**

**}**

Don’t forget to setup a makefile for your project. It will look something like this but will need to be changed based on your project.

OBJECTS = source/loader.o \

source/kmain.o \ source/io.o \ drivers/frame\_buffer.o \

drivers/hardware\_interrupt\_enabler.o \ drivers/interrupt\_asm.o \ drivers/interrupt\_handlers.o \ drivers/interrupts.o \ drivers/keyboard.o \

drivers/pic.o

CC = gcc

CFLAGS = -I. -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \

-nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c LDFLAGS = -T ./source/link.ld -melf\_i386

AS = nasm ASFLAGS = -f elf all: kernel.elf

kernel.elf: $(OBJECTS)

ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf

cp kernel.elf iso/boot/kernel.elf genisoimage -R

-b boot/grub/stage2\_eltorito

-no-emul-boot

-boot-load-size 4

-A os

-input-charset utf8

-quiet

-boot-info-table

-o os.iso iso

\

\

\

\

\

\

\

\

\

run: os.iso

qemu-system-i386 -nographic -boot d -cdrom os.iso -m 32 -d cpu -D

↪ logQ.txt

run-curses: os.iso

qemu-system-i386 -curses \

-monitor telnet::75454,server,nowait \

-chardev stdio,id=char0 \

-serial chardev:char0 \

-boot d \

-cdrom os.iso \

-m 32 \

-d cpu \

-no-reboot \

-no-shutdown \

-D logQ.txt

%.o: %.c

$(CC) $(CFLAGS) $< -o $@

%.o: %.s

$(AS) $(ASFLAGS) $< -o $@

clean:

rm -rf \*.o source/\*.o drivers/\*.o kernel.elf os.iso ios/boot/kernel.elf

# Tasks

## Part 1 ‑ Display Keyboard Input

Extend the interrupt\_handler function in **interrupts.c** to process keyboard input. Your imple‑ mentation should:

1. Read the scan code from the keyboard
2. Convert it to ASCII
3. Handle special cases:
   * Backspace: Remove last character
   * Enter: Move to new line
   * Regular characters: Display on screen
4. Clear screen when appropriate Example addition to interrupt\_handler:

**case** INTERRUPTS\_KEYBOARD**:**

**while ((**inb**(**0x64**) &** 1**)) {**

input **=** keyboard\_read\_scan\_code**();**

**if (!(**input **&** 0x80**)) {** *// Key press, not release* ascii **=** keyboard\_scan\_code\_to\_ascii**(**input**); if (**ascii **==** '\b'**) {**

*// Handle backspace*

fb\_backspace**();**

**} else if (**ascii **==** '\n'**) {**

*// Handle newline*

fb\_newline**();**

**} else if (**ascii **!=** 0**) {**

*// Display character*

fb\_write\_char**(**ascii**);**

**}**

**}**

**}**

pic\_acknowledge**(**interrupt**); break;**

## Part 2 ‑ Input Buffer API

Create a system to efficiently store and retrieve keyboard input. You need to implement two key func‑ tions:

1. getc() Function Requirements:
   * Remove and return a single character from the buffer
   * Handle empty buffer conditions appropriately
   * Consider using a circular buffer to manage input efficiently
   * Must be callable from other parts of your OS
2. readline() Function Requirements:
   * Read characters until a newline is encountered
   * Use your getc() function to retrieve characters
   * Store the result in a provided buffer
   * Handle buffer size limits safely
   * Return when enter/newline is detected

Think carefully about: ‑ Buffer size limitations ‑ How to handle buffer overflow ‑ Synchronization be‑ tween interrupt handler and buffer access ‑ Efficient memory usage ‑ Error conditions

## Part 3 ‑ Terminal Implementation

Create a basic terminal interface that processes user commands. Your terminal should feel similar to a basic Unix shell.

Requirements:

1. Command Processing
   * Display a prompt (e.g., “myos>”)
   * Accept user input until enter is pressed
   * Parse input into command and arguments
   * Execute appropriate function based on command
   * Handle unknown commands gracefully
2. Suggested Commands (Pick at least 2)
   * echo [text] ‑ Display the provided text
   * clear ‑ Clear the screen
   * help ‑ Show available commands
   * version ‑ Display OS version
   * shutdown ‑ Prepare system for shutdown
3. Implementation Hints:
   * Split your input string at space characters to separate command and arguments
   * Consider using a command structure like:

**struct** command **{**

const char**\*** name**;**

void **(\***function**)(**char**\*** args**);**

**};**

* + Store commands in an array or table for easy lookup
  + Remember to null‑terminate your strings
  + Handle empty input gracefully

1. Command Parsing Steps:
   * Identify the command (first word)
   * Separate arguments (remaining text)
   * Look up command in your command table
   * Pass arguments to appropriate handler
   * Display results

Think about: ‑ How to handle command line editing ‑ Error messages for unknown commands ‑ Com‑ mand output formatting ‑ Memory management ‑ Input string validation ## Extension Tasks

If you complete the main tasks, try these extensions: 1. Add command history (accessible with up/down arrows) 2. Implement tab completion for commands 3. Add color support to the terminal 4. Create a basic file system command (ls) ## Common Issues

* Keyboard not responding: Check PIC initialization
* Characters not displaying: Verify framebuffer code
* System hanging: Check interrupt acknowledgment
* Buffer overflow: Verify array bounds
* Missing characters: Check scan code conversion

# Resources

* OS Dev Wiki: https://wiki.osdev.org/
* OS Handbook
* Previous worksheet materials