Chapter 6

Interrupts and Input

Now that the OS can produce *output* it would be nice if it also could get some *input*. (The operating system must be able to handle *interrupts* in order to read information from the keyboard). An interrupt occurs when a hardware device, such as the keyboard, the serial port or the timer, signals the CPU that the state of the device has changed. The CPU itself can also send interrupts due to program errors, for example when a program references memory it doesn't have access to, or when a program divides a number by zero. Finally, there are also *software intterupts*, which are interrupts that are caused by the `int` assembly code instruction, and they are often used for system calls.

Interrupts Handlers

Interrupts are handled via the *Interrupt Descriptor Table* (IDT). The IDT describes a handler for each interrupt. The interrupts are numbered (0 - 255) and the handler for interrupt i is defined at the *ith* position in the table. There are three different kinds of handlers for interrupts:

- Task handler
- Interrupt handler
- Trap handler

The task handlers use functionality specific to the Intel version of x86, so they won't be covered here (see the Intel manual [33], chapter 6, for more info). The only difference between an interrupt handler and a trap handler is that the interrupt handler disables interrupts, which means you cannot get an interrupt while at the same time handling an interrupt. In this book, we will use trap handlers and disable interrupts manually when we need to.

Creating an Entry in the IDT

An entry in the IDT for an interrupt handler consists of 64 bits. The highest 32 bits are shown in the figure below:

```
Bit:     | 31              16 | 15 | 14 13 | 12 | 11 | 10 9 8 | 7 6 5 | 4 3 2 1 0 |
Content: | offset high        | P  | DPL   | 0  | D  | 1  1 0 | 0 0 0 | reserved  |
```

The lowest 32 bits are presented in the following figure:

```
Bit:     | 31              16 | 15              0 |
Content: | segment selector   | offset low          |
```

A description for each name can be found in the table below:

Name	Description
offset high	The 16 highest bits of the 32 bit address in the segment.
offset low	The 16 lowest bits of the 32 bits address in the segment.
p	If the handler is present in memory or not (1 = present, 0 = not present).
DPL	Descriptor Privilige Level, the privilege level the handler can be called from (0, 1, 2, 3).
D	Size of gate, (1 = 32 bits, 0 = 16 bits).
segment selector	The offset in the GDT.
r	Reserved.

The offset is a pointer to code (preferably an assembly code label). For example, to create an entry for a handler whose code starts at `0xDEADBEEF` and that runs in privilege level 0 (therefore using the same code segment selector as the kernel) the following two bytes would be used:

```
0xDEAD8E00
0x0008BEEF
```

If the IDT is represented as an `unsigned integer idt[512]` then to register the above example as an handler for interrupt 0 (divide-by-zero), the following code would be used:

```
idt[0] = 0xDEAD8E00
idt[1] = 0x0008BEEF
```

As written in the chapter “Getting to C”, we recommend that you instead of using bytes (or unsigned integers) use packed structures to make the code more readable.

Handling an Interrupt

When an interrupt occurs the CPU will push some information about the interrupt onto the stack, then look up the appropriate interrupt hander in the IDT and jump to it. The stack at the time of the interrupt will look like the following:

```
[esp + 12] eflags
[esp + 8]  cs
[esp + 4]  eip
[esp]      error code?
```

The reason for the question mark behind error code is that not all interrupts create an *error code*. The specific CPU interrupts that put an error code on the stack are 8, 10, 11, 12, 13, 14 and 17. The error code can be used by the interrupt handler to get more information on what has happened. Also, note that the interrupt *number* is *not* pushed onto the stack. We can only determine what interrupt has occurred by knowing what code is executing - if the handler registered for interrupt 17 is executing, then interrupt 17 has occurred.

Once the interrupt handler is done, it uses the `iret` instruction to return. The instruction `iret` expects the stack to be the same as at the time of the interrupt (see the figure above). Therefore, any values pushed onto the stack by the interrupt handler must be popped. Before returning, `iret` restores `eflags` by popping the value from the stack and then finally jumps to `cs:eip` as specified by the values on the stack.

The interrupt handler has to be written in assembly code, since all registers that the interrupt handlers use must be preserved by pushing them onto the stack. This is because the code that was interrupted doesn't know about the interrupt and will therefore expect that its registers stay the same. Writing all the logic of the interrupt handler in assembly code will be tiresome. Creating a handler in assembly code that saves the registers, calls a C function, restores the registers and finally executes `iret` is a good idea!

The C handler should get the state of the registers, the state of the stack and the number of the interrupt as arguments. The following definitions can for example be used:

```
struct cpu_state {
    unsigned int eax;
    unsigned int ebx;
    unsigned int ecx;
    .
    .
    .
    unsigned int esp;
} __attribute__((packed));

struct stack_state {
    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
} __attribute__((packed));

void interrupt_handler(struct cpu_state cpu, struct stack_state stack, unsigned int interrupt);
```

Creating a Generic Interrupt Handler

Since the CPU does not push the interrupt number on the stack it is a little tricky to write a generic interrupt handler. This section will use macros to show how it can be done. Writing one version for each interrupt is tedious - it is better to use the macro functionality of NASM [34]. And since not all interrupts produce an error code the value 0 will be added as the "error code" for interrupts without an error code. The following code shows an example of how this can be done:

```
%macro no_error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword 0                     ; push 0 as error code
```

```
    push    dword %1                    ; push the interrupt number
    jmp     common_interrupt_handler    ; jump to the common handler
%endmacro

%macro error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword %1                    ; push the interrupt number
    jmp     common_interrupt_handler    ; jump to the common handler
%endmacro

common_interrupt_handler:               ; the common parts of the generic interrupt handler
    ; save the registers
    push    eax
    push    ebx
    .
    .
    .
    push    ebp

    ; call the C function
    call    interrupt_handler

    ; restore the registers
    pop     ebp
    .
    .
    .
    pop     ebx
    pop     eax

    ; restore the esp
    add     esp, 8

    ; return to the code that got interrupted
    iret

no_error_code_interrupt_handler 0       ; create handler for interrupt 0
no_error_code_interrupt_handler 1       ; create handler for interrupt 1
.
.
.
error_code_handler              7       ; create handler for interrupt 7
.
.
.
```

The `common_interrupt_handler` does the following:

- Push the registers on the stack.
- Call the C function `interrupt_handler`.
- Pop the registers from the stack.

- Add 8 to `esp` (because of the error code and the interrupt number pushed earlier).
- Execute `iret` to return to the interrupted code.

Since the macros declare global labels the addresses of the interrupt handlers can be accessed from C or assembly code when creating the IDT.

Loading the IDT

The IDT is loaded with the `lidt` assembly code instruction which takes the address of the first element in the table. It is easiest to wrap this instruction and use it from C:

```
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]    ; load the address of the IDT into register eax
    lidt    eax             ; load the IDT
    ret                     ; return to the calling function
```

Programmable Interrupt Controller (PIC)

To start using hardware interrupts you must first configure the Programmable Interrupt Controller (PIC). The PIC makes it possible to map signals from the hardware to interrupts. The reasons for configuring the PIC are:

- Remap the interrupts. The PIC uses interrupts 0 - 15 for hardware interrupts by default, which conflicts with the CPU interrupts. Therefore the PIC interrupts must be remapped to another interval.
- Select which interrupts to receive. You probably don't want to receive interrupts from all devices since you don't have code that handles these interrupts anyway.
- Set up the correct mode for the PIC.

In the beginning there was only one PIC (PIC 1) and eight interrupts. As more hardware were added, 8 interrupts were too few. The solution chosen was to chain on another PIC (PIC 2) on the first PIC (see interrupt 2 on PIC 1).

The hardware interrupts are shown in the table below:

PIC 1	Hardware	PIC 2	Hardware
0	Timer	8	Real Time Clock
1	Keyboard	9	General I/O
2	PIC 2	10	General I/O
3	COM 2	11	General I/O
4	COM 1	12	General I/O

PIC 1	Hardware	PIC 2	Hardware
5	LPT 2	13	Coprocessor
6	Floppy disk	14	IDE Bus
7	LPT 1	15	IDE Bus

A great tutorial for configuring the PIC can be found at the SigOPS website [35]. We won't repeat that information here.

Every interrupt from the PIC has to be acknowledged - that is, sending a message to the PIC confirming that the interrupt has been handled. If this isn't done the PIC won't generate any more interrupts.

Acknowledging a PIC interrupt is done by sending the byte `0x20` to the PIC that raised the interrupt. Implementing a `pic_acknowledge` function can thus be done as follows:

```
#include "io.h"

#define PIC1_PORT_A 0x20
#define PIC2_PORT_A 0xA0

/* The PIC interrupts have been remapped */
#define PIC1_START_INTERRUPT 0x20
#define PIC2_START_INTERRUPT 0x28
#define PIC2_END_INTERRUPT   PIC2_START_INTERRUPT + 7

#define PIC_ACK     0x20

/** pic_acknowledge:
 *  Acknowledges an interrupt from either PIC 1 or PIC 2.
 *
 *  @param num The number of the interrupt
 */
void pic_acknowledge(unsigned integer interrupt)
{
    if (interrupt < PIC1_START_INTERRUPT || interrupt > PIC2_END_INTERRUPT) {
      return;
    }

    if (interrupt < PIC2_START_INTERRUPT) {
      outb(PIC1_PORT_A, PIC_ACK);
    } else {
      outb(PIC2_PORT_A, PIC_ACK);
    }
}
```

Reading Input from the Keyboard

The keyboard does not generate ASCII characters, it generates scan codes. A scan code represents a button - both presses and releases. The scan code representing the just pressed button can be read from the keyboard's data I/O port which has address `0x60`. How this can be done is shown in the following example:

```
#include "io.h"

#define KBD_DATA_PORT   0x60

/** read_scan_code:
 *  Reads a scan code from the keyboard
 *
 *  @return The scan code (NOT an ASCII character!)
 */
unsigned char read_scan_code(void)
{
    return inb(KBD_DATA_PORT);
}
```

The next step is to write a function that translates a scan code to the corresponding ASCII character. If you want to map the scan codes to ASCII characters as is done on an American keyboard then Andries Brouwer has a great tutorial [36].

Remember, since the keyboard interrupt is raised by the PIC, you must call `pic_acknowledge` at the end of the keyboard interrupt handler. Also, the keyboard will not send you any more interrupts until you read the scan code from the keyboard.

Further Reading

- The OSDev wiki has a great page on interrupts, http://wiki.osdev.org/Interrupts
- Chapter 6 of Intel Manual 3a [33] describes everything there is to know about interrupts.