Handling Interrupts in the ULIX Operating System

(Literate Version)

Hans-Georg Eßer

May 11, 2015

Contents

List of Figures

List of Tables

1 Introduction

This document contains a selected chapter from the introductory operating systems book "The Design and Implementation of the Ulix Operating System" [EF15]. In this excerpt we only treat the handling of interrupts and faults on an Intel-i386-based machine.

The Ulix source code consists of one large LaTeX file with embedded code, following the Literate Programming style [Knu84] from which two files `ulix.c` (C code) and `start.asm` (Assembler code) are automatically generated using the `noweb` tool [Ram94]. Those in turn are compiled or assembled and the resulting object files are linked to create the kernel binary.

Initialization of the interrupt and fault handlers occurs early in the system's overall initialization process.

In the following chapter you will find some code chunks which are continuations of earlier chunk definitions (starting in other chapters that are not part of this excerpt):

- ⟨*constants* 6⟩ contains definitions of (macro) constants which are declared via the `#define` pre-processor statement.
- ⟨*macro definitions* 21c⟩ are also `#define`'d macros, but "real" ones which use parameters and thus resemble functions.
- ⟨*type definitions* 11a⟩ is the place for all structure definitions.
- ⟨*global variables* 12a⟩ collects all variables which are visible throughout the whole Ulix system.
- ⟨*function prototypes* 7a⟩ contains prototypes of C functions, and
- ⟨*function implementations* 7b⟩ the implementations of these functions.

The general structure of the main source file, `ulix.c` is the following:

[2a] ⟨*ulix.c* 2a⟩≡
⟨*copyright notice* ⟩
⟨*constants* 6⟩ ⟨*public constants* ⟩
⟨*macro definitions* 21c⟩ ⟨*public macro definitions* ⟩
⟨*public elementary type definitions* ⟩
⟨*type definitions* 11a⟩ ⟨*public type definitions* 16a⟩
⟨*function prototypes* 7a⟩ ⟨*public function prototypes* ⟩
⟨*global variables* 12a⟩
⟨*function implementations* 7b⟩ ⟨*public function implementations* ⟩
⟨*kernel main* 2b⟩

This chunk is extracted to the `ulix.c` source code file. The last past of this chunk, ⟨*kernel main* 2b⟩ contains the implementation of the kernel's `main`$_{2b}$`()` function:

[2b] ⟨*kernel main* 2b⟩≡ (2a)
```
void main () {
  ⟨initialize kernel global variables ⟩
  ⟨setup serial port ⟩  // for debugging
  ⟨setup memory ⟩
  ⟨setup video ⟩
  ⟨setup keyboard ⟩
  ⟨initialize system 2c⟩
  ⟨initialize syscalls ⟩
  ⟨initialize filesystem ⟩
  ⟨initialize swap ⟩
  initialize_module ();  // external code
  ⟨start init process ⟩
}
```

and in there you can find the reference to ⟨*initialize system* 2c⟩ which uses further code chunks

[2c] ⟨*initialize system* 2c⟩≡ (2b)
⟨*install the interrupt descriptor table* 20d⟩
⟨*install the fault handlers* 21d⟩
⟨*install the interrupt handlers* 13a⟩
⟨*install the timer* ⟩
⟨*enable interrupts* 25b⟩

most of which are shown in this excerpt of the Ulix book. Assembler source code is collected in the ⟨*start.asm* 18⟩ chunk which gets extracted to `start.asm`.

All code chunks which appear without a page number (for example: ⟨*install the timer* ⟩ from the last chunk) are defined outside this document; you can look at the Ulix book to see the full Ulix code.

2
Interrupts and Faults

All modern CPUs and even many of the older ones such as the Zilog Z80 8-bit processor can be interrupted: the CPU has an input line which can be triggered by an external device connected to this line. When such an interrupt occurs, the current activity is suspended, and the CPU continues operation at a specified address: it executes an interrupt handler.

In principle a device could be directly connected to the CPU, but modern machines contain many devices which want to interrupt the processor, e. g. the disk controllers, the keyboard controller, the serial ports, or the on-board clock. Thus an extra device, called the *interrupt controller*, intermediates between the other devices and the CPU. One of the advantages of such an interrupt controller is that it is programmable: it is possible to enable or disable specific interrupts whereas the CPU itself can only completely enable or disable all interrupts, using the `sti` (set interrupt flag) and `cli` (clear interrupt flag) instructions. (These machine instructions exist on Intel-x86-compatible CPUs; other chips have similar instructions.) Being programmable also means that interrupt numbers can be remapped (we will see later why this is helpful). Interrupt controllers with these features are called *programmable interrupt controllers* (PICs), and we'll use that abbreviation throughout the rest of this chapter.

interrupt controller

`sti`, `cli`

PICs

After the implementation of interrupts we will also take a look at fault handling since the involved mechanisms are very similar to those which we need for handling interrupts. The main difference between interrupts and faults is that faults occur as a direct consequence of some specific instruction that our code executes. In that sense they are *synchronous*. Interrupts on the other hand occur without any connection to the currently executing instruction, since they are not triggered (immediately) by our code but by some device. That is why they are called *asynchronous*.

synchronous

asynchronous

2.1 Examples for Interrupt Usage

Interrupt handling is a core functionality which is used in lots of places: without interrupts we would not be able to build a useful operating system.

Let's look at some example features of Ulix which depend heavily on interrupts:

timer handler

Multi-tasking Ulix can execute several processes in parallel and switch between them using a simple round-robin scheduling mechanism. That is only possible because the clock chip on the motherboard regularly generates timer interrupts, and Ulix installs a timer interrupt handler which—when activated—calls the scheduler to check whether it is time to switch to a different process. If there were no interrupts, we could only implement *non-preemptive scheduling* which relies on the processes to give up the CPU voluntarily.

keyboard handler

Keyboard input Whenever you press or release a key on a PC, either event generates an interrupt. Ulix picks up these interrupts and the keyboard interrupt handler reads a key press or key release code from the controller.

polling

A keyboard driver does not need interrupts, but the alternative is to constantly poll (query) the keyboard controller in order to find out whether a new event has occurred. That's possible but wastes a lot of CPU time. Polling does not work well in a multi-tasking environment. (However for a single-tasking operating system it may be good enough.)

ATA/FDC handlers

Media Reading and writing hard disks and floppy disks also depends on interrupts: In the Ulix implementation of filesystems (and disk access) a process which wants to read or write makes a system call which sends a request to the drive controller. Then Ulix puts the calling process to sleep. Once the request has been served, the drive controller generates an interrupt, and the interrupt handler for the hard disk controller or the floppy disk controller (these are two separate handlers) deals with the data and wakes up the sleeping process.

Again, this could be done without interrupts. But the process would have to remain active and continuously poll the controller to find out whether the data transfer has been completed.

Serial ports Finally, the serial ports are similar to the keyboard, since all of them are *character devices*: they transfer single bytes (instead of blocks of bytes).

2.2 Interrupt Handling on the Intel Architecture

Intel 8259 PIC

The classical IBM PC used the Intel 8259 Programmable Interrupt Controller, compatible descendents of which are still used in modern computers. The 8259 has eight input lines (through which up to eight separate devices may connect) and one output line which forwards received interrupt signals to the CPU. It is possible to use more than one 8259 PIC

since these controllers can be cascaded which means that a second controller's output pin is connected with one of the first controller's input pins (typically the one for device 2, see Figure 2.1). With that cascade, devices connected to the first controller keep their normal numbers (0, 1, 3–7 with 2 reserved for the second controller), and devices connected to the second controller use device numbers between 8 and 15, allowing for a total of 15 (= 16 − 1) separate device numbers. The first or primary controller is called *Master PIC*, the second one is the *Slave PIC* (as it is not directly connected to the CPU but relies on the master PIC to have its interrupts signals forwarded). The numbers 0–15 are called *Interrupt Request Numbers* (IRQs).

Cascading PICs

Master PIC

Slave PIC

IRQ

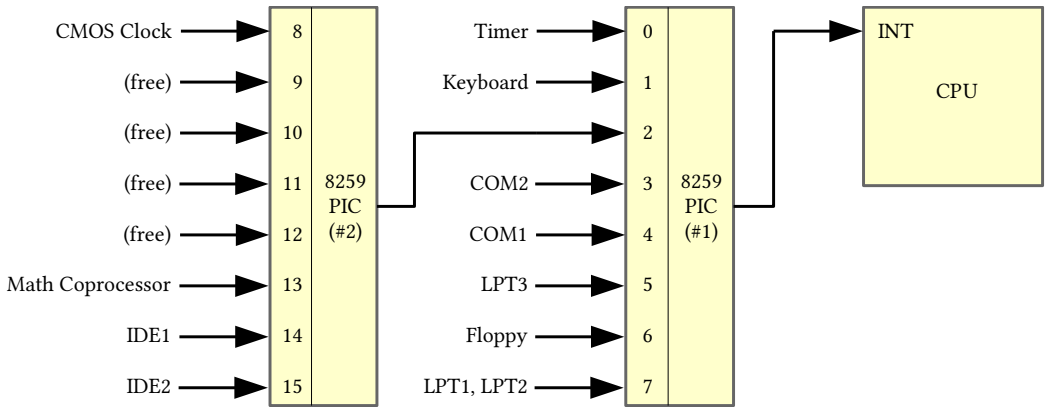

Figure 2.1: Two PICs are cascaded, which allows for 15 distinct interrupts.

As you can see from the figure, there is a fixed mapping of some devices to specific IRQs. We will use the following IRQs in the Ulix implementation:

- **0: Timer Chip**. On a PC's mainboard you can find a (programmable) timer chip which regularly generates interrupts. We will use timer interrupts to call the scheduler (besides other tasks).
- **1: Keyboard**. This is the interrupt generated by PS/2 keyboards. A USB keyboard would be handled differently, but we do not support USB devices.
- **2: Slave PIC**. As already mentioned, IRQ 2 is reserved for connecting the secondary (slave) PIC.
- **3: Serial Port 2**. The second serial port will be used for our implementation of what we've called the *serial hard disk*—it is discussed in the full Ulix book.
- **4: Serial Port 1**. We only use the first serial port for output (when running Ulix in a PC emulator), thus we will not install an interrupt handler for this IRQ.
- **6: Floppy**. This is the IRQ for the floppy controller. It can handle up to two floppy drives.
- **14: Primary IDE Controller**. And finally, 14 is the IRQ of the primary IDE controller. Many PC mainboards contain two controllers, with each of them allowing two drives to connect. The secondary IDE controller would generate the interrupt number 15,

but we're going to support only one controller.

We can define names for the IRQ numbers right now:

[6] ⟨*constants* 6⟩≡ (2a) 8▷

```
#define IRQ_TIMER      0
#define IRQ_KBD        1
#define IRQ_SLAVE      2     // Here the slave PIC connects to master
#define IRQ_COM2       3
#define IRQ_COM1       4
#define IRQ_FDC        6
#define IRQ_IDE       14     // primary IDE controller; secondary has IRQ 15
```

Defines:
IRQ_SLAVE, used in chunk 13a.

2.2.1 Using Ports for I/O Requests

Going where? We want to initialize the PICs, which means directly talking to these controllers. Like with most other devices we can use the machine instructions `in` and `out` to find out the PIC's current status and tell it what to do.

Here we provide the code which lets us access the controllers.

I/O Ports Access to many hardware components (including the PICs) is possible via *I/O ports*. Using `in` and `out` machine instructions it is possible to transfer bytes, words or doublewords between a CPU register and a memory location or register on some device (such as a hard disk controller).

The Intel 80386 Programmer's Reference Manual [Int86, pp. 146–147] explains:

`in`, `out`

> "The I/O instructions `IN` and `OUT` are provided to move data between I/O ports and the *EAX* (32-bit I/O), the *AX* (16-bit I/O) or *AL* (8-bit I/O) general registers. `IN` and `OUT` instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the *DX* register to one of up to 64K port addresses.
>
> `IN` (Input from Port) transfers a byte, word or doubleword from an input port to *AL*, *AX* or *EAX*. If a program specifies *AL* with the `IN` instruction, the processor transfers 8 bits from the selected port to *AL*. If a program specifies *AX* with the `IN` instruction, the processor transfers 16 bits from the port to *AX*. If a program specifies *EAX* with the `IN` instruction, the processor transfers 32 bits from the port to *EAX*.
>
> `OUT` (Output to Port) transfers a byte, word or doubleword to an output port from *AL*, *AX* or *EAX*. The program can specify the number of the port using the same methods as the `IN` instruction."

For accessing 8-bit, 16-bit and 32-bit ports, the Intel assembler language provides separate commands `inb` / `outb` (byte), `inw` / `outw` (word) and `inl` / `outl` (long: doubleword) which make it explicit what kind of transfer is wanted. We'll use them in the functions

⟨*function prototypes* 7a⟩≡ (2a) 12b▷ [7a]

```
byte inportb (word port);
word inportw (word port);
void outportb (word port, byte data);
void outportw (word port, word data);
```

There are several possible C implementations with inline assembler code, the following code is most readable:

⟨*function implementations* 7b⟩≡ (2a) 12c▷ [7b]

```
byte inportb (word port) {
  byte retval;  asm volatile ("inb %%dx, %%al" : "=a"(retval) : "d"(port));
  return retval;
}

word inportw (word port) {
  word retval;  asm volatile ("inw %%dx, %%ax" : "=a" (retval) : "d" (port));
  return retval;
}

void outportb (word port, byte data) {
  asm volatile ("outb %%al, %%dx" : : "d" (port), "a" (data));
}

void outportw (word port, word data) {
  asm volatile ("outw %%ax, %%dx" : : "d" (port), "a" (data));
}
```

Defines:
`inportb`, used in chunk 13f.
`outportb`, used in chunks 9, 13d, and 20a.
`outportw`, used in chunk 7a.

We could provide `inportl` and `outportl` (for 32-bit values) in a similar fashion, using `inl`, `outl` and the 32-bit register *EAX* (instead of the 16-bit and 8-bit versions *AX* and *AL*), but we do not need them. (Remember that *EAX*, *AX* and *AL* are (parts of) the same register, see Figure 2.2. On a 64-bit machine, *RAX* is the 64-bit extended version of *EAX*.)

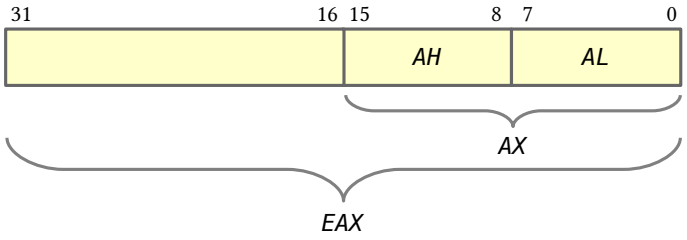

Figure 2.2: The lower half of *EAX* is *AX* which in turn is split into *AH* (high) and *AL* (low).

2.2.2 Initializing the PIC

Going where?

Now that we have functions for talking to devices we can set up the two PICs. We will configure one as master and the other as slave, and we also remap the interrupt numbers from 0–15 to 32–47 because the first 32 numbers are reserved for faults (see Section 2.3).

The PICs can be accessed via the following four ports:

[8] ⟨*constants* 6⟩+≡ (2a) ◁6 19a▷

```
// I/O Addresses of the two programmable interrupt controllers
#define IO_PIC_MASTER_CMD   0x20  // Master (IRQs 0-7), command register
#define IO_PIC_MASTER_DATA  0x21  // Master, control register

#define IO_PIC_SLAVE_CMD    0xA0  // Slave (IRQs 8-15), command register
#define IO_PIC_SLAVE_DATA   0xA1  // Slave, control register
```

Defines:
IO_PIC_MASTER_CMD, used in chunks 9a and 20a.
IO_PIC_MASTER_DATA, used in chunks 9 and 13.
IO_PIC_SLAVE_CMD, used in chunks 9a and 20a.
IO_PIC_SLAVE_DATA, used in chunks 9 and 13.

They need to be initialized by sending them four "Initialization Command Words" (ICW) called ICW1, ICW2, ICW3 and ICW4 in a specific order, using specific ports. Each of the PICs has a command register and a data register. During normal operation we can write to the data register (using the ports `IO_PIC_MASTER_DATA`$_8$ and `IO_PIC_SLAVE_DATA`$_8$ for PIC1 or PIC2, respectively) to set the *interrupt mask*: That's a byte where each bit tells the controller whether it shall respond to a specific interrupt (1 means: mask, i. e., ignore the interrupt; 0 means: forward it to the CPU). We will start with an interrupt mask of `0xFF` for each controller (all bits are 1), thus all hardware interrupts will be ignored.

interrupt mask

The following code was taken from Bran's Kernel Development Tutorial [Fri05] (e. g. from the source file `irq.c`) and modified.

For programming the controller, we can send configuration data to the data port, but we have to initialize the programming by writing to the command port. The complete sequence is as follows:

- First we send ICW1 to both PICs. ICW1 is a byte whose bits have the following meaning [Int88, p. 11]:

  0 D_0: ICW4 needed? We set this to 1 since we want to program the controller.

  1 D_1: Single (1) / Cascade (0) mode: We set this to 0 since there's a slave.

  2 D_2: Call Address Interval (ignored), the default value is 0.

  3 D_3: Level (1) / Edge (0) Triggered Mode: we set this to 0.

  4 D_4: Initialization Bit: We set it to 1 because we want to initialize the controller.

  5,6,7 D_5, D_6, D_7: not used on x86 hardware, set to 0.

  This results in the byte `00010001` (`0x11`). The value is the same for both PICs. As mentioned before, ICW1 must be sent to the PICs' command registers.

⟨*remap the interrupts to 32..47* 9a⟩≡ (13a) 9b▷ [9a]
```
outportb (IO_PIC_MASTER_CMD,  0x11); // ICW1: initialize; begin programming
outportb (IO_PIC_SLAVE_CMD,   0x11); // ICW1: dito, for PIC2
```
Uses IO_PIC_MASTER_CMD 8, IO_PIC_SLAVE_CMD 8, and outportb 7b.

- In the next step we send ICW2 to the PICs' data registers. The lowest three bits specify the offset for remapping the interrupts. Since the first 32 interrupts must be reserved for processor exception handlers (e. g. "Division by Zero" and "Page Fault" handlers), we map the interrupts 0–15 to the range 32–47 (`0x20` – `0x2f`).

  remap the interrupts

  Each PIC would normally generate interrupts in the range 0–7, thus the offset is not the same for both PICs: For PIC1 it is `0x20` (32; mapping 0–7 to 32-39), and for PIC2 it is `0x28` (40; mapping 0–7 to 40–47).

⟨*remap the interrupts to 32..47* 9a⟩+≡ (13a) ◁9a 9c▷ [9b]
```
outportb (IO_PIC_MASTER_DATA, 0x20); // ICW2 for PIC1: offset 0x20
                                     // (remaps 0x00..0x07 -> 0x20..0x27)
outportb (IO_PIC_SLAVE_DATA,  0x28); // ICW2 for PIC2: offset 0x28
                                     // (remaps 0x08..0x0f -> 0x28..0x2f)
```
Uses IO_PIC_MASTER_DATA 8, IO_PIC_SLAVE_DATA 8, and outportb 7b.

- The next command word is ICW3. Its functionality depends on whether we send it to the master (PIC1) or the slave (PIC2): The PICs already know that they are master and slave (because we sent that information as part of ICW1) [Int88, p. 12].

  The master expects a command word byte in which each set bit specifies a slave connected to it. We have only one slave and want to make it signal new interrupts on interrupt line 2 of the master. Thus, only the third bit (from the right) must be set: $\texttt{00000100}_b$ = `0x04`.

  The slave needs a slave ID. We give it the ID 2 = `0x02`.

⟨*remap the interrupts to 32..47* 9a⟩+≡ (13a) ◁9b 9d▷ [9c]
```
outportb (IO_PIC_MASTER_DATA, 0x04); // ICW3 for PIC1: there's a slave on IRQ 2
                                     // (0b00000100 = 0x04)
outportb (IO_PIC_SLAVE_DATA,  0x02); // ICW3 for PIC2: your slave ID is 2
```
Uses IO_PIC_MASTER_DATA 8, IO_PIC_SLAVE_DATA 8, and outportb 7b.

- To end the sequence, we send ICW4 which is just `0x01` for x86 processors [Int88, p. 12].

⟨*remap the interrupts to 32..47* 9a⟩+≡ (13a) ◁9c [9d]
```
outportb (IO_PIC_MASTER_DATA, 0x01); // ICW4 for PIC1 and PIC2: 8086 mode
outportb (IO_PIC_SLAVE_DATA,  0x01);
```
Uses IO_PIC_MASTER_DATA 8, IO_PIC_SLAVE_DATA 8, and outportb 7b.

With the remapping in place we can now create entries for the interrupt handler table—we need some new data structures for them.

2.2.3 Interrupt Descriptor Table

Going where? The PICs are initialized and will do the right thing when an interrupt occurs, but we haven't told the CPU yet what to do when it receives one. This calls for a new data structure, the *Interrupt Descriptor Table*, which we must define according to the Intel standards and fill with proper values.

While the first Intel-8086/8088-based personal computers used a fixed address in RAM to store the interrupt handler addresses, modern machines let us place the table anywhere in memory. After preparing the table we must use the machine instruction `lidt` (load interrupt descriptor table register) to tell the CPU where to search.

`lidt`

The procedure we need to follow is similar to the one for activating segmentation via a GDT (Global Descriptor Table; see full Ulix book):

1. We first store interrupt descriptors (each of which is eight bytes large) in a table consisting of `struct idt_entry`$_{11a}$ entries,
2. then we create some kind of pointer structure `struct idt_ptr`$_{11b}$ which contains the length and the start address of the table,
3. and finally we execute `lidt` (compare this to `lgdt` which loads the GDT).

Figure 2.3 shows the layout of an IDT entry. The *Flags* halfbyte (second line, left in the figure) consists of

- the present flag (bit 3) which must always be set to 1,
- two bits (2 and 1) for the Descriptor Privilege Level (DPL). We will always set this to $11_b = 3$ since we want all interrupts to be available all the time (when we're in kernel or user mode) and
- a so-called "storage segment" flag (bit 0; which must be set to 0 for an "interrupt gate", see next entry).

The *Type* halfbyte declares what kind of descriptor this is: we will always set it to 1110_b, making this descriptor an

- *80386 32-bit interrupt gate* descriptor (which is what we want).

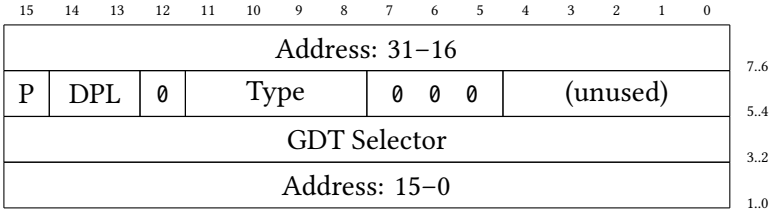

Figure 2.3: An Interrupt Descriptor contains the address of an interrupt handler and some configuration information.

Besides this type, there are alternatives:

- 0101_b for an *80386 32-bit task gate*,
- 0110_b for an *80286 16-bit interrupt gate*,
- 0111_b for an *80286 16-bit trap gate* and
- 1111_b for an *80386 32-bit trap gate*,

but we will not go into the details about these. Instead of interrupt gates we could also use trap gates, the difference between those being that "for interrupt gates, interrupts are automatically disabled upon entry and reenabled upon IRET which restores the saved *EFLAGS*" [OSD13].

We will use the following datatype definitions:

⟨ *type definitions* 11a ⟩≡ (2a) 11b▷ [11a]

```
struct idt_entry {
    unsigned int addr_low  : 16;   // lower 16 bits of address
    unsigned int gdtsel    : 16;   // use which GDT entry?
    unsigned int zeroes    :  8;   // must be set to 0
    unsigned int type      :  4;   // type of descriptor
    unsigned int flags     :  4;
    unsigned int addr_high : 16;   // higher 16 bits of address
} __attribute__((packed));
```

Defines:
idt_entry, used in chunks 12a and 20d.

The *selector* must be the number of a code segment descriptor (in the GDT); we will always set this to 0x08 since our kernel (ring 0) code segment uses that number (see code chunk ⟨ *install flat gdt* ⟩ which is only available in the full Ulix book).

The IDT pointer has the same structure as the GDT pointer: it informs about the length and the location of the IDT:

⟨ *type definitions* 11a ⟩+≡ (2a) ◁11a [11b]

```
struct idt_ptr {
    unsigned int limit   : 16;
    unsigned int base    : 32;
} __attribute__((packed));
```

Defines:
idt_ptr, used in chunk 12a.

In theory, an interrupt number can be any byte, i. e., a value between 0 and 255. We will use a full IDT with 256 entries even though most of the entries will be null descriptors—if somehow an interrupt is generated which has a null descriptor, the CPU will generate an "unhandled interrupt" exception. We will talk about exceptions right after we've finished the interrupt handling code.

[12a] ⟨*global variables* 12a⟩≡ (2a) 12e▷
```
struct idt_entry idt[256] = { { 0 } };
struct idt_ptr idtp;
```
Defines:
idt, used in chunks 12c and 20d.
idtp, used in chunks 20d and 21a.
Uses idt_entry 11a and idt_ptr 11b.

The variables $\texttt{idt}_{12a}$ and $\texttt{idtp}_{12a}$ will now be used in a way that is similar to how we used `gdt` (a `struct gdt_entry[]` array) and `gp` (a `struct gdt_ptr` structure) when we wrote the GDT code (again, this information is only available in the ULIX book).

We start with a function

[12b] ⟨*function prototypes* 7a⟩+≡ (2a) ◁7a 12d▷
```
void fill_idt_entry (byte num, unsigned long address,
    word gdtsel, byte flags, byte type);
```

which writes an entry of the IDT:

[12c] ⟨*function implementations* 7b⟩+≡ (2a) ◁7b 13d▷
```
void fill_idt_entry (byte num, unsigned long address,
    word gdtsel, byte flags, byte type) {
  if (num ≥ 0 && num < 256) {
    idt[num].addr_low  = address & 0xFFFF; // address is the handler address
    idt[num].addr_high = (address >> 16) & 0xFFFF;
    idt[num].gdtsel    = gdtsel;           // GDT sel.: user mode or kernel mode?
    idt[num].zeroes    = 0;
    idt[num].flags     = flags;
    idt[num].type      = type;
  }
}
```
Defines:
fill_idt_entry, used in chunks 12b, 13a, and 21c.
Uses idt 12a.

Parts of all of our interrupt handlers will be assembler code (which we store in `start.asm`); we'll explain soon why that has to be. For the moment, let's declare 16 external function symbols $\texttt{irq0}_{18}$, $\texttt{irq1}_{18}$, ..., $\texttt{irq15}_{18}$ whose addresses we're about to enter into the IDT with $\texttt{fill\_idt\_entry}_{12c}$:

[12d] ⟨*function prototypes* 7a⟩+≡ (2a) ◁12b 13b▷
```
extern void irq0(), irq1(), irq2(),  irq3(),  irq4(),  irq5(),  irq6(),  irq7();
extern void irq8(), irq9(), irq10(), irq11(), irq12(), irq13(), irq14(), irq15();
```

We will store the function addresses in an array which simplifies accessing them:

[12e] ⟨*global variables* 12a⟩+≡ (2a) ◁12a 19b▷
```
void (*irqs[16])() = {
  irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7,      // store them in
  irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15      // an array
};
```

Defines:
irqs, used in chunk 13a.
Uses irq0 18, irq1 18, irq10 18, irq11 18, irq12 18, irq13 18, irq14 18, irq15 18, irq2 18, irq3 18, irq4 18, irq5 18, irq6 18, irq7 18, irq8 18, and irq9 18.

The following code chunk enters their address in the IDT:

⟨*install the interrupt handlers* 13a⟩≡ (2c) [13a]
```
⟨remap the interrupts to 32..47 9a⟩
set_irqmask (0xFFFF);           // initialize IRQ mask
enable_interrupt (IRQ_SLAVE);   // IRQ slave

for (int i = 0;  i < 16;  i++) {
  fill_idt_entry (32 + i,
                  (unsigned int)irqs[i],
                  0x08,
                  0b1110,     // flags: 1 (present), 11 (DPL 3), 0
                  0b1110);    // type:  1110 (32 bit interrupt gate)
}
```
Uses enable_interrupt 14, fill_idt_entry 12c, IRQ_SLAVE 6, irqs 12e, and set_irqmask 13d.

This code chunk sets the *IRQ mask* to 0xFFFF = 1111111111111111_b via IRQ mask

⟨*function prototypes* 7a⟩+≡ (2a) ◁12d 13c▷ [13b]
```
static void set_irqmask (word mask);
```

which disables all interrupts, and then it enables the interrupt for the slave PIC with

⟨*function prototypes* 7a⟩+≡ (2a) ◁13b 13e▷ [13c]
```
static void enable_interrupt (int number);
```

—both functions have not been mentioned so far. The IRQ mask is a 16-bit value in which each bit says whether some interrupt is enabled (value 0) or not (value 1). We must talk to both PICs to set the mask, the master PIC gets the lower eight bits (for the interrupts 0–7), the slave PIC gets the upper eight bits (for the interrupts 8–15):

⟨*function implementations* 7b⟩+≡ (2a) ◁12c 13f▷ [13d]
```
static void set_irqmask (word mask) {
  outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
  outportb (IO_PIC_SLAVE_DATA,  (char)(mask >> 8)  );
}
```
Defines:
set_irqmask, used in chunks 13 and 14.
Uses IO_PIC_MASTER_DATA 8, IO_PIC_SLAVE_DATA 8, and outportb 7b.

We can also read the mask from the two PICs with a similar function we call

⟨*function prototypes* 7a⟩+≡ (2a) ◁13c 20b▷ [13e]
```
word get_irqmask ();
```

in which we read the two data registers instead of writing them:

⟨*function implementations* 7b⟩+≡ (2a) ◁13d 14▷ [13f]
```
word get_irqmask () {
  return inportb (IO_PIC_MASTER_DATA)
      + (inportb (IO_PIC_SLAVE_DATA) << 8);
}
```

Defines:
get_irqmask, used in chunks 13e and 14.
Uses inportb 7b, IO_PIC_MASTER_DATA 8, and IO_PIC_SLAVE_DATA 8.

In other chapters (of the Ulix book) we will often enable a specific interrupt for some device after we've prepared its usage, e. g. for the floppy controller. For that purpose, we will always use enable_interrupt$_{14}$() like we did above. It simply reads the current IRQ mask, clears a bit, and writes the new value back:

[14] ⟨*function implementations* 7b⟩+≡ (2a) ◁13f 20a▷

```
static void enable_interrupt (int number) {
  set_irqmask (
    get_irqmask ()        // the current value
    & ~(1 << number)      // 16 one-bits, but bit "number" cleared
  );
}
```

Defines:
enable_interrupt, used in chunk 13.
Uses get_irqmask 13f and set_irqmask 13d.

2.2.4 Writing the Interrupt Handler

Going where? Everything is prepared for interrupt handlers — now we need to define them, i. e., implement the irq0$_{18}$(), ... irq15$_{18}$() functions. This step requires some assembler code and some C code.

We have installed handlers for all 16 interrupts, but what do they do? We will define part of their code in the assembler file, but we start with a description of what we expect to happen in general.

When an interrupt occurs, the CPU suspends the currently running code, saves some information on the stack, and then jumps to the address that it finds in the IDT. (It also uses a different stack and switches to kernel mode if it was in user mode when the interrupt occurred.) Then the interrupt handler runs, and once it has finished its job, it returns with the iret instruction. iret is different from the regular ret instruction which normal functions use for returning to the calling function: it is the special "return from interrupt" instruction which restores the original state (user or kernel mode, stack, *EFLAGS* register) so that the regular code can continue as if the interrupt had never happened.

iret

Switching to the interrupt handler can mean a change of the privilege level that the CPU executes in: So far we've only let Ulix work in ring 0 (kernel mode), but later when we introduce processes (in the Ulix book) it can happen that an interrupt occurs while the CPU runs in ring 3 (user mode). If that is the case, the privilege level changes (from 3 to 0). When such a transition occurs, the information (return address etc.) is not written to the process' user mode stack, but on the process' kernel stack which is located elsewhere and normally used during the execution of *system calls*. For now, the relevant piece of information is that different information gets stored on the "target stack": In case of a privilege change the CPU first writes the contents of the *SS* and *ESP* registers on the (new)

stack—this does not happen if the CPU was already operating in ring 0. Next, *EFLAGS*, *CS* and *EIP* are written to the stack: that is all we need for returning to the interrupted code. Figure 2.4 shows the different stack contents when the interrupt handler starts executing [Int86, p. 159].

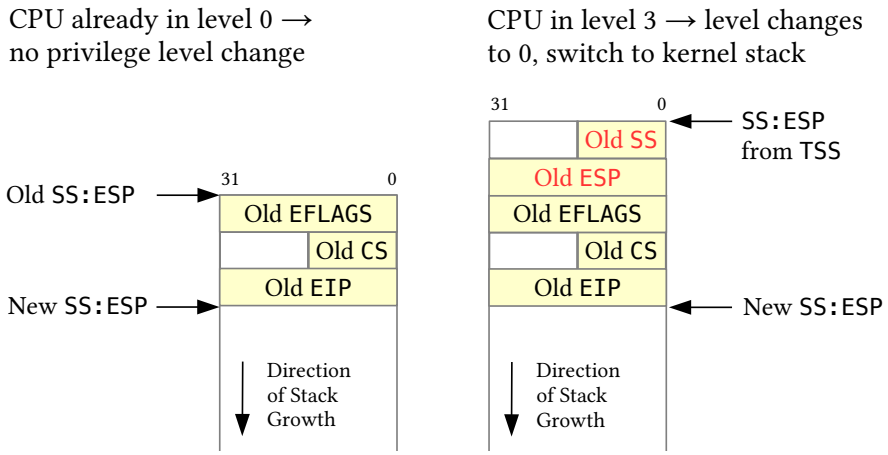

Figure 2.4: When entering the interrupt handler, the stack contains information for returning from the handler. Left: without privilege level change; right: with change from level 3 to 0, extra data marked red.

We cannot directly use a C function as an interrupt handler because once it would finish its work, it would do a regular `RET` which does not do what we want. (Of course we could use inline assembler code inside the C function to make it work anyway, but it makes more sense to directly implement parts of the handlers in assembler.)

2.2.4.1 The Context Data Structure

We want to be able to define handler functions in C which get called from the assembler code. Those functions will all have the following prototype:

```
void handler_function (context_t *r);
```

where $\texttt{context\_t}_{16a}$ is a central data structure that can hold all the registers we use on the Intel machine. It will also be used in fault handlers, system call handlers and several other functions which need information about the current state. context

We define the $\texttt{context\_t}_{16a}$ structure so that it matches the way in which we set up the stack in the assembler part of the handler:

[16a] ⟨*public type definitions* 16a⟩≡ (2a)

```
typedef struct {
  unsigned int gs, fs, es, ds;
  unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
  unsigned int int_no, err_code;
  unsigned int eip, cs, eflags, useresp, ss;
} context_t;
```

Defines:
context_t, used in chunks 16b, 20, and 25a.

2.2.4.2 Assembler Part of the Handler

In order to have a handler function see useful values in the structure that `r` points to, we need to push the register contents in the reverse order onto the stack:

[16b] ⟨*push registers onto the stack* 16b⟩≡ (17b 18 24b)

```
pusha
push ds
push es
push fs
push gs
push esp  ; pointer to the context_t
```

The first instruction `pusha` (push all general registers) pushes a lot of registers onto the stack: *EAX*, *ECX*, *EDX*, *EBX*, the old value of *ESP* (before the `pusha` execution began), *EBP*, *ESI*, and *EDI*—in that order. We add the segment registers *DS*, *ES*, *FS* and *GS*, and you can see that we've successfully handled the first two lines of the $\texttt{context\_t}_{16a}$ type definition. When the interrupt occurred, the registers *EFLAGS*, *CS* and *EIP* (and possibly also *SS* and the user mode's *ESP*) were also pushed on the stack which gives us the values in the fourth line of the $\texttt{context\_t}_{16a}$ definition.

What's missing are the values on the third line: We want to tell the handler *which interrupt* occurred so that we can use the same interrupt handler for several interrupts—for example, if we supported both IDE controllers (with interrupts 14 and 15) we could use that trick to run the same IDE handler when either of those interrupts occurred; thus, between the automatically happening push operations and the ones we perform in ⟨*push registers onto the stack* 16b⟩ we also push the interrupt number and another value `err_code` which can hold an error code. Interrupts don't have an error code, but we will recycle the same code later when we deal with faults, and some of those do provide an error code.

The final `push esp` statement in ⟨*push registers onto the stack* 16b⟩ is necessary because we cannot just place the structure contents on the stack: the handler function expects a pointer ($\texttt{context\_t}_{16a}$ `*r`), and `ESP` contains just that pointer: the start address of the structure. Figure 2.5 shows the layout of the stack after the assembler part has finished the preparations.

Later, when the handler's task is completed, we will need to pop the registers from the stack—in the reverse order:

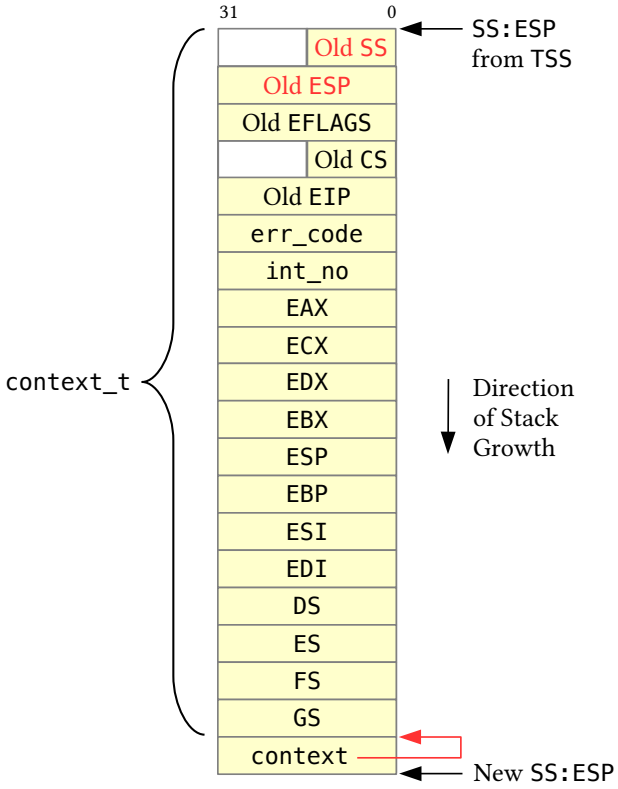

Figure 2.5: Stack after interrupt handler initialization by the assembler part.

⟨*pop registers from the stack* 17a⟩≡ (17b 18 24b) [17a]

```
pop esp
pop gs
pop fs
pop es
pop ds
popa
```

Now here's an example of how we could implement the interrupt handler for IRQ 15:

⟨*irq15 example* 17b⟩≡ [17b]

```
push byte 0           ; error code
push byte 15          ; interrupt number
⟨push registers onto the stack 16b⟩
call irq_handler      ; call C function
⟨pop registers from the stack 17a⟩
add esp, 8            ; for errcode, irq no.
iret
```

Uses irq_handler 20a.

This contains all we need:

1. The two `push` commands add the error code and the interrupt number (which is 15 in this example).
2. With ⟨*push registers onto the stack* 16b⟩ we complete the `context_t`$_{16a}$ data structure and also push a pointer to it.
3. Now the stack is prepared properly to call the C function `irq_handler`$_{20a}$.
4. After returning, we first have to undo the push operations with ⟨*pop registers from the stack* 17a⟩.
5. Then we modify the stack address: we add 8, thus undoing the two `push` operations for the error code and the interrupt number.
6. Finally we return from the handler with `iret`.

We need almost the same code 16 times (for IRQs 0 to 15)—the only difference between the 16 versions is the interrupt number that we push in the second instruction. We simplify our code by having our individual handlers just push the two values (0 and the interrupt number) and then jump to an address which provides the common commands. The 0 value is a placeholder for an error code which cannot occur in interrupt handlers, but (as mentioned before) we will also implement fault handlers which shall use the same stack layout, and some of them will write a fault-specific error code into that location.

[18] ⟨*start.asm* 18⟩≡ 21a▷

```
global irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7
global irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15

%macro irq_macro 1
       push byte 0          ; error code (none)
       push byte %1         ; interrupt number
       jmp irq_common_stub  ; rest is identical for all handlers
%endmacro

irq0:  irq_macro 32
irq1:  irq_macro 33
irq2:  irq_macro 34
irq3:  irq_macro 35
irq4:  irq_macro 36
irq5:  irq_macro 37
irq6:  irq_macro 38
irq7:  irq_macro 39
irq8:  irq_macro 40
irq9:  irq_macro 41
irq10: irq_macro 42
irq11: irq_macro 43
irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
irq15: irq_macro 47
```

```
extern irq_handler          ; defined in the C source file

irq_common_stub:            ; this is the identical part
        ⟨push registers onto the stack 16b⟩
        call irq_handler    ; call C function
        ⟨pop registers from the stack 17a⟩
        add esp, 8
        iret
```

Defines:
irq0, used in chunk 12e.
irq1, used in chunk 12e.
irq10, used in chunk 12e.
irq11, used in chunk 12e.
irq12, used in chunk 12e.
irq13, used in chunk 12e.
irq14, used in chunk 12e.
irq15, used in chunk 12e.
irq2, used in chunk 12e.
irq3, used in chunk 12e.
irq4, used in chunk 12e.
irq5, used in chunk 12e.
irq6, used in chunk 12e.
irq7, used in chunk 12e.
irq8, used in chunk 12e.
irq9, used in chunk 12.
Uses irq_handler 20a.

Our interrupt handling code is a slightly improved version of the code which Bran's Kernel Tutorial [Fri05] uses; the original code contains some extra instructions that we don't need for the ULIX kernel.

2.2.4.3 C Part of the Handler

Finally, we show what happens when the assembler code calls the external handler function irq_handler$_{20a}$() that we implement in the C file.

The first thing our handler needs to do is acknowledge the interrupt. For that purpose it sends the command

⟨*constants* 6⟩+≡ (2a) ◁8 [19a]

```
#define END_OF_INTERRUPT  0x20
```

Defines:
END_OF_INTERRUPT, used in chunk 20a.

to all PICs which are involved: In case of an interrupt number between 0 and 7 that is only the primary PIC; in case the number is 8 or higher, both controllers need to be informed. Omitting this step would stop the controller from raising further interrupts which would basically disable interrupts completely.

Next we check whether a specific handler for the current interrupt has been installed in the

⟨*global variables* 12a⟩+≡ (2a) ◁12e 24c▷ [19b]

```
void *interrupt_handlers[16] = { 0 };
```

Defines:
interrupt_handlers, used in chunk 20.

array of interrupt handlers.

[20a] ⟨*function implementations* 7b⟩+≡ (2a) ◁14 20c▷

```
void irq_handler (context_t *r) {
  int number = r->int_no - 32;                    // interrupt number
  void (*handler)(context_t *r);                  // type of handler functions

  if (number ≥ 8) {
    outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT); // notify slave  PIC
  }
  outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);  // notify master PIC (always)

  handler = interrupt_handlers[number];
  if (handler != NULL)  handler (r);
}
```

Defines:
`irq_handler`, used in chunks 17b and 18.
Uses `context_t` 16a, `END_OF_INTERRUPT` 19a, `interrupt_handlers` 19b, `IO_PIC_MASTER_CMD` 8, `IO_PIC_SLAVE_CMD` 8, and `outportb` 7b.

As a last step we provide a function

[20b] ⟨*function prototypes* 7a⟩+≡ (2a) ◁13e 20e▷

```
void install_interrupt_handler (int irq, void (*handler)(context_t *r));
```

Uses `context_t` 16a and `install_interrupt_handler` 20c.

which lets us enter (pointers to) handler functions in this array; it is pretty simple:

[20c] ⟨*function implementations* 7b⟩+≡ (2a) ◁20a 25a▷

```
void install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
  if (irq ≥ 0 && irq < 16)
    interrupt_handlers[irq] = handler;
}
```

Defines:
`install_interrupt_handler`, used in chunk 20b.
Uses `context_t` 16a and `interrupt_handlers` 19b.

Early in the ⟨*initialize system* 2c⟩ step of the kernel's $\texttt{main}_{2b}\texttt{()}$ function (which is not shown in this excerpt) we need to load the Interrupt Descriptor Table Register (*IDTR*) so that the CPU can find the table:

IDTR

[20d] ⟨*install the interrupt descriptor table* 20d⟩≡ (2c)

```
idtp.limit = (sizeof (struct idt_entry) * 256) - 1;   // must do -1
idtp.base  = (int) &idt;
idt_load ();
```

Uses `idt` 12a, `idt_entry` 11a, `idt_load` 21a, and `idtp` 12a.

It uses the assembler function

[20e] ⟨*function prototypes* 7a⟩+≡ (2a) ◁20b 21b▷

```
extern void idt_load ();
```

lidt

which writes the address of $\texttt{idtp}_{12a}$ to the *IDTR* register via the `lidt` instruction:

⟨ *start.asm* 18 ⟩+≡ ◁18 22a▷ [21a]

```
extern idtp               ; defined in the C file
global idt_load
idt_load:    lidt [idtp]
             ret
```

Defines:
idt_load, used in chunk 20.
Uses idtp 12a.

In other chapters of the ULIX book we will often use this function in commands similar to

```
install_interrupt_handler (IRQ_SOMEDEV, somedev_handler);
```

2.3 Faults

As we've mentioned in the introduction to this chapter, handling a fault is very similar to handling an interrupt. Since you've just seen the interrupt code, you will recognize many concepts at once while we present the fault handling code.

Like we defined the interrupt handlers $\texttt{irq0}_{18}\texttt{()}$ to $\texttt{irq15}_{18}\texttt{()}$ in the assembler file `start.asm`, we do the same with 32 fault handler functions $\texttt{fault0}_{22c}\texttt{()}$ to $\texttt{fault31}_{22c}\texttt{()}$.

⟨ *function prototypes* 7a ⟩+≡ (2a) ◁20e 24d▷ [21b]

```
extern void
  fault0(),  fault1(),  fault2(),  fault3(),  fault4(),  fault5(),  fault6(),
  fault7(),  fault8(),  fault9(),  fault10(), fault11(), fault12(), fault13(),
  fault14(), fault15(), fault16(), fault17(), fault18(), fault19(), fault20(),
  fault21(), fault22(), fault23(), fault24(), fault25(), fault26(), fault27(),
  fault28(), fault29(), fault30(), fault31(), fault128();
```

Uses fault0 22c, fault1 22c, fault10 22c, fault11 22c, fault12 22c, fault13 22c, fault14 22c, fault15 22c, fault16 22c, fault17 22c, fault18 22c, fault19 22c, fault2 22c, fault20 22c, fault21 22c, fault22 22c, fault23 22c, fault24 22c, fault25 22c, fault26 22c, fault27 22c, fault28 22c, fault29 22c, fault3 22c, fault30 22c, fault31 22c, fault4 22c, fault5 22c, fault6 22c, fault7 22c, fault8 22c, and fault9 22c.

and we enter these in the IDT just like we did with the `irq*()` functions. Since there are so many of them, we'll use a macro that calls $\texttt{fill\_idt\_entry}_{12c}\texttt{()}$ with the same *flags* and *type* arguments as before:

⟨ *macro definitions* 21c ⟩≡ (2a) [21c]

```
#define FILL_IDT(i) fill_idt_entry (i, (unsigned int)fault##i, 0x08, 0b1110, 0b1110)
```

Defines:
FILL_IDT, used in chunk 21d.
Uses fill_idt_entry 12c.

⟨ *install the fault handlers* 21d ⟩≡ (2c) [21d]

```
FILL_IDT( 0); FILL_IDT( 1); FILL_IDT( 2); FILL_IDT( 3); FILL_IDT( 4); FILL_IDT( 5);
FILL_IDT( 6); FILL_IDT( 7); FILL_IDT( 8); FILL_IDT( 9); FILL_IDT(10); FILL_IDT(11);
FILL_IDT(12); FILL_IDT(13); FILL_IDT(14); FILL_IDT(15); FILL_IDT(16); FILL_IDT(17);
FILL_IDT(18); FILL_IDT(19); FILL_IDT(20); FILL_IDT(21); FILL_IDT(22); FILL_IDT(23);
FILL_IDT(24); FILL_IDT(25); FILL_IDT(26); FILL_IDT(27); FILL_IDT(28); FILL_IDT(29);
FILL_IDT(30); FILL_IDT(31); FILL_IDT(128);
```

Uses FILL_IDT 21c.

After macro expansion this will generate commands such as

```
fill_idt_entry (31,(unsigned)fault31, 0x08, 0b1110, 0b1110);
```

In the assembler file we use the same trick for the `fault*()` functions that you've just seen for `irq*()`:

[22a] ⟨*start.asm* 18⟩+≡ ◁21a 22b▷

```
global fault0,  fault1,  fault2,  fault3,  fault4,  fault5,  fault6,  fault7
global fault8,  fault9,  fault10, fault11, fault12, fault13, fault14, fault15
global fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23
global fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31
```

The handlers all look similar: We push one or two bytes on the stack and then jump to `fault_common_stub`$_{24b}$. The choice of one or two arguments depends on the kind of interrupt that occurred: for some faults the CPU pushes a one-byte error code on the stack, and for some others it does not. In order to have the same stack setup (regardless of the fault) we push an extra null byte in those cases where no error code is pushed.

error code

The code always looks like one of the following two cases:

```
fault5:  push byte 0  ; error code
         push byte 5
         jmp fault_common_stub
```

```
fault8:  ; no error code
         push byte 8
         jmp fault_common_stub
```

Since we do not want to type this repeatedly, we use `nasm`'s macro feature which lets us write simple macros for both cases. `fault_macro_0`$_{22b}$ handles the cases where we need to push an extra null byte (as in `fault5`$_{22c}$ above), and `fault_macro_no0`$_{22b}$ handles the other cases (as in `fault8`$_{22c}$ above):

[22b] ⟨*start.asm* 18⟩+≡ ◁22a 22c▷

```
%macro fault_macro_0 1
        push byte 0  ; error code
        push byte %1
        jmp fault_common_stub
%endmacro
```

```
%macro fault_macro_no0 1
        ;  don't push error code
        push byte %1
        jmp fault_common_stub
%endmacro
```

Defines:
`fault_macro_0`, used in chunk 22c.
`fault_macro_no0`, used in chunk 22c.
Uses `fault_common_stub` 24b.

With these macros the rest is straight-forward:

[22c] ⟨*start.asm* 18⟩+≡ ◁22b 24a▷

```
fault0:  fault_macro_0     0   ;  Divide by Zero
fault1:  fault_macro_0     1   ;  Debug
fault2:  fault_macro_0     2   ;  Non Maskable Interrupt
fault3:  fault_macro_0     3   ;  INT 3
fault4:  fault_macro_0     4   ;  INTO
fault5:  fault_macro_0     5   ;  Out of Bounds
fault6:  fault_macro_0     6   ;  Invalid Opcode
```

```
fault7:  fault_macro_0     7   ;  Coprocessor not available
fault8:  fault_macro_no0   8   ;  Double Fault
fault9:  fault_macro_0     9   ;  Coprocessor Segment Overrun
fault10: fault_macro_no0  10   ;  Bad TSS
fault11: fault_macro_no0  11   ;  Segment Not Present
fault12: fault_macro_no0  12   ;  Stack Fault
fault13: fault_macro_no0  13   ;  General Protection Fault
fault14: fault_macro_no0  14   ;  Page Fault
fault15: fault_macro_0    15   ;  (reserved)
fault16: fault_macro_0    16   ;  Floating Point
fault17: fault_macro_0    17   ;  Alignment Check
fault18: fault_macro_0    18   ;  Machine Check
fault19: fault_macro_0    19   ;  (reserved)
fault20: fault_macro_0    20   ;  (reserved)
fault21: fault_macro_0    21   ;  (reserved)
fault22: fault_macro_0    22   ;  (reserved)
fault23: fault_macro_0    23   ;  (reserved)
fault24: fault_macro_0    24   ;  (reserved)
fault25: fault_macro_0    25   ;  (reserved)
fault26: fault_macro_0    26   ;  (reserved)
fault27: fault_macro_0    27   ;  (reserved)
fault28: fault_macro_0    28   ;  (reserved)
fault29: fault_macro_0    29   ;  (reserved)
fault30: fault_macro_0    30   ;  (reserved)
fault31: fault_macro_0    31   ;  (reserved)
```

Defines:
fault0, used in chunk 21b.
fault1, used in chunk 21b.
fault10, used in chunk 21b.
fault11, used in chunk 21b.
fault12, used in chunk 21b.
fault13, used in chunk 21b.
fault14, used in chunk 21b.
fault15, used in chunk 21b.
fault16, used in chunk 21b.
fault17, used in chunk 21b.
fault18, used in chunk 21b.
fault19, used in chunk 21b.
fault2, used in chunk 21b.
fault20, used in chunk 21b.
fault21, used in chunk 21b.
fault22, used in chunk 21b.
fault23, used in chunk 21b.
fault24, used in chunk 21b.
fault25, used in chunk 21b.
fault26, used in chunk 21b.
fault27, used in chunk 21b.
fault28, used in chunk 21b.
fault29, used in chunk 21b.
fault3, used in chunk 21b.
fault30, used in chunk 21b.
fault31, used in chunk 21b.
fault4, used in chunk 21b.
fault5, used in chunk 21b.
fault6, used in chunk 21b.

`fault7`, used in chunk 21b.
`fault8`, used in chunk 21b.
`fault9`, used in chunks 21b and 22a.
Uses `fault_macro_0` 22b and `fault_macro_no0` 22b.

`fault_common_stub`$_{24b}$ is—almost—a rewrite of `irq_common_stub`$_{18}$, the only difference is that we call a different C function `fault_handler`$_{25a}$`()` in the middle.

[24a] ⟨*start.asm* 18⟩+≡ ◁22c 24b▷

```
extern fault_handler
```

Uses `fault_handler` 25a.

The stub saves the processor state, calls the handler function and restores the stack frame:

[24b] ⟨*start.asm* 18⟩+≡ ◁24a

```
fault_common_stub:
    ⟨push registers onto the stack 16b⟩
    call fault_handler    ; call C function
    ⟨pop registers from the stack 17a⟩
    add esp, 8            ; for errcode, irq no.
    iret
```

Defines:
`fault_common_stub`, used in chunk 22b.
Uses `fault_handler` 25a.

Initially our fault handlers will just output a message stating the cause of the fault and then halt the system; later we will provide fault handlers for some types of faults which try to solve the problem and let the operation go on. Here are the error messages:

[24c] ⟨*global variables* 12a⟩+≡ (2a) ◁19b

```
char *exception_messages[] = {
    "Division By Zero",        "Debug",                          //  0,  1
    "Non Maskable Interrupt",  "Breakpoint",                     //  2,  3
    "Into Detected Overflow",  "Out of Bounds",                  //  4,  5
    "Invalid Opcode",          "No Coprocessor",                 //  6,  7
    "Double Fault",            "Coprocessor Segment Overrun",    //  8,  9
    "Bad TSS",                 "Segment Not Present",            // 10, 11
    "Stack Fault",             "General Protection Fault",       // 12, 13
    "Page Fault",              "Unknown Interrupt",              // 14, 15
    "Coprocessor Fault",       "Alignment Check",                // 16, 17
    "Machine Check",                                             // 18
    "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
    "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
    "Reserved", "Reserved", "Reserved"                           // 19..31
};
```

Defines:
`exception_messages`, used in chunk 26a.

We get the correct message by accessing the proper entry of the array, e. g., for a page fault (with fault number 14) it is stored in `exception_messages`$_{24c}$`[14]`.

Fault Handler

Our C fault handler

[24d] ⟨*function prototypes* 7a⟩+≡ (2a) ◁21b

```
void fault_handler (context_t *r);
```

displays some information about the problem and checks whether the fault occurred while a process was running (by testing whether `r->eip < 0xc0000000`). If not, the system switches to the kernel mode shell (and is broken).

The page fault handler is a special case: we expect to deal with page faults silently (see the page fault chapter of the ULIX book), so we check for this case before doing anything else.

⟨*function implementations* 7b⟩+≡ (2a) ◁20c [25a]

```
void fault_handler (context_t *r) {
  if (r->int_no == 14) {                          // fault 14 is a page fault
    page_fault_handler (r); return;
  }

  memaddress fault_address = (memaddress)(r->eip);

  if (r->int_no < 32) {
    ⟨fault handler: display status information 26a⟩

    if ( fault_address < 0xc0000000 ) {           // user mode
      ⟨fault handler: terminate process 26b⟩
    }

    ⟨disable scheduler ⟩                          // error inside the kernel
    ⟨enable interrupts 25b⟩
    printf ("\n");
    asm ("jmp kernel_shell");
  }
}
```

Defines:
`fault_handler`, used in chunk 24.
Uses `context_t` 16a and `printf`.

For completely enabling or disabling interrupts we use the `sti` and `cli` Assembler instructions that we mentioned earlier in this chapter:

⟨*enable interrupts* 25b⟩≡ (2c 25a) [25b]

```
asm ("sti");    // set interrupt flag
```

⟨*disable interrupts* 25c⟩≡ [25c]

```
asm ("cli");    // clear interrupt flag
```

The `fault_handler`$_{25a}$ function also contains a reference to a code chunk named ⟨*disable scheduler* ⟩ which belongs to the scheduler implementation and cannot be discussed in this excerpt.

For displaying the status information we look at the register contents which are provided by `r`. Especially interesting are the task number, the address space number, the address of the faulting instruction, the *EFLAGS* register and the error code which the CPU has provided upon entry into the fault handler.

[26a] ⟨*fault handler: display status information* 26a⟩≡ (25a)

```
printf ("'%s' Exception at 0x%08x (task=%d, as=%d).\n",
  exception_messages[r->int_no], r->eip, current_task, current_as);
printf ("eflags: 0x%08x  errcode: 0x%08x\n", r->eflags, r->err_code);
printf ("eax: %08x  ebx: %08x  ecx: %08x  edx: %08x \n",
  r->eax, r->ebx, r->ecx, r->edx);
printf ("eip: %08x  esp: %08x  int: %8d  err: %8d \n",
  r->eip, r->esp, r->int_no, r->err_code);
printf ("ebp: %08x  cs: 0x%02x ds: 0x%02x es: 0x%02x fs: 0x%02x ss: 0x%02x \n",
  r->ebp, r->cs, r->ds, r->es, r->fs, r->ss);
printf ("User mode stack: 0x%08x-0x%08x\n", TOP_OF_USER_MODE_STACK
  - address_spaces[current_as].stacksize, TOP_OF_USER_MODE_STACK);
```

Uses `exception_messages` 24c and `printf`.

If a process was running, the fault handler terminates it:

[26b] ⟨*fault handler: terminate process* 26b⟩≡ (25a)

```
mutex_lock (thread_list_lock);
  thread_table[current_task].state = TSTATE_ZOMBIE;
  remove_from_ready_queue (current_task);
mutex_unlock (thread_list_lock);
r->ebx = -1;  // exit_ code for this process
syscall_exit (r);
```

Since we have not talked about processes yet, you need not worry about the `mutex_lock` and `mutex_unlock` commands (which protect the thread table) as well as the other references to the thread table via `thread_table[current_task]` or `remove_from_ready_queue()`. We will explain all these functions and the thread table data structure later, and we will also show what the `syscall_exit()` function does. You can choose to ignore the complete ⟨*fault handler: terminate process* 26b⟩ block in the code for now.

A page fault need not be a problem: it often occurs because the code attempted to access an invalid address (which is bad), but yet more often the address will be valid, but the page won't be in the physical RAM. That situation can be helped. The ULIX book describes the implementation of the *page fault handler*. It requires a working hard disk since we will *page out* pages to the disk and later *page them in* again.

page fault handler

Chunk Index

Identifier Index

Index

K

L

M

N

O

P

R

S

T

Z

Bibliography

[EF15] Hans-Georg Eßer and Felix C. Freiling. *The Design and Implementation of the Ulix Operating System*. 2015. To be published.

[Fri05] Brandon Friesen. Bran's Kernel Development. A tutorial on writing kernels. Version 1.0, 2005. http://www.osdever.net/bkerndev/Docs/title.htm; accessed 2013/10/19.

[Int86] Intel. Intel 80386 Programmer's Reference Manual, 1986. http://microsym.com/editor/assets/386intel.pdf, accessed 2013/11/17.

[Int88] Intel. Intel 8259A Programmable Interrupt Controller (8259A/8259A-2), December 1988. Datasheet for PC interrupt controller chip, http://i30www.ira.uka.de/~sdi/doc/8259a.pdf.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[OSD13] OSDev.org. Interrupt Descriptor Table, July 2013. http://wiki.osdev.org/Interrupt_Descriptor_Table, last accessed 2013/11/16.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.