Computer Labs 2013/2014 - 1st Semester
Lab 3: The PC's Keyboard

---

1. Objectives

The objectives of this lab are threefold:

1. that you learn the operation of the PC's keyboard and how to use its low level interface.
2. that you learn how to mix C-code and ASM code in your program.
3. that you get a better understanding of the interrupt mechanism and that you learn how to use it in the context of the PC's keyboard

This lab is planned for 2 lab classes. You are expected to attend the lectures and in addition to spend about 5 hours to prepare each of the lab classes.

This handout is structured in the following sections:

2. What to Do

Write in C language several functions that use the PC's keyboard low level interface. The key functionality to implement is:

1. Read and display the scancodes, both the make codes and the break codes, generated by the PC's keyboard;
2. To change the state of the indicator LEDs of the PCs keyboard

Furthermore, we want you to learn:

1. How to mix C-code with assembly code
2. How to handle interrupts from more than one I/O device

Unlike in previous labs you are not given the prototypes of the functions to implement: the specification of these functions is part of your job. However, to make the task of grading your assignment feasible, you are required to implement the following test functions:

```
1. int kbd_test_scan(unsigned short asm)
2. int kbd_test_leds(unsigned short n, unsigned short *toggle)
3. int kbd_test_timed_scan(unsigned short n)
```

These functions are declared in header file `test3.h`, and source file `test3.c` contains their implementation stubs. You may find it convenient to add your test code to that file; this way you will avoid mistakes in their

definition. Section 5 describes what these functions should do.

2.1 Class Preparation

This lab is planned for two lab classes.

First class

The goals for the first class is to implement:

1. `int kbd_test_scan()`, with the IH written in C
2. `int kbd_test_leds(unsigned short n, unsigned short *toggle)`

Therefore, for the first class you should read the material presented in the lecture of October 14th about the PC's keyboard and this handout except Sections 4.3, 4.4 and 5.3, which are for the second class.

Furthermore, you should:

1. Create a folder named `lab3` at the top level of your SVN repository
2. Create a file named `lab3.c` with the `main()` function that allows to test your code in a flexible way without having to compile it. You should also add this file, and the test files provided to your SVN repository.

You may wish to read the information provided in Section 8 about some SVN commands that may be useful to perform these actions.

Second class

The goals for the second class is to implement:

1. `int kbd_test_scan()`, with the IH written in assembly
2. `int kbd_test_timed_scan(unsigned short n)`

Therefore, for the second class you should read also Sections 4.3, 4.4 and 5.3 of this handout, and the material presented in the lecture of October 14th about assembly programming.

Furthermore you should also refresh your knowledge of IA32 assembly programming.

3. The PC Keyboard and its Controller

Whenever a user presses or releases one of its keys, the keyboard sends a **scancode** to the KBC. The KBC puts that code in a register, and generates an interrupt, if configured to do so. It is then up to the interrupt handler to read the scancode from the KBC. Although the KBC can be used in polled mode, this is seldom the case.

3.1 Scancodes

A key's scancode depends only on the position of that key in the keyboard. It is different from the ASCII code (or other common encoding) with which it is labeled. The conversion between the key's scancodes and the code of the character on that key is usually done by the operating system using a keymap. For example, by default Minix 3 uses the US keyboard keymap. The Minix 3 images you are using in LCOM use a "handcrafted" Portuguese keymap, because Minix 3 does not include any in its distribution. The advantage of this approach is clear for keyboard manufacturers: a key in a given position, always generates the same scandcode, independently of whether the keyboard uses the US layout or another layout.

Furthermore, the scancode generated when a key is pressed is different from the scancode generated when that key is released. To distinguish between them, we call the former **make code** and the latter **break code**, a

commonly used terminology. Usually, the break code of a key differs from the make code of that key in that the MSB of a break code is set whereas that of the make code is not. For example, the make code of the `ESC` key is `0x01` whereas its break code is `0x81`.

Using different make and break codes provides a lot of flexibility. For example, a keyboard needs not generate a different scancode for a key when the `Shift` key is pressed. Rather, it generates a make code for `Shift` and it is then up to the keyboard driver to do the necessary mapping between the reception of the `Shift` make code and the reception of its break code.

Most PC's scancodes are one byte long, although some special keys have longer scan codes. Two-byte long scancodes usually use `0xE0` as their first byte. This prefix is used in both the make and the break codes. Again, the difference between the make and the break codes of a key is in the MSB, now of the second byte of the scancode. Some keyboards have keys that generate even longer scancodes, but you need not worry about it in LCOM.

The scancodes that we have described are known as **Set 1**, that is are the set of scancodes of the PC-XT. Another set of scancodes, known as **Set 2** appeared with the PC-AT and is widely used. However, the KBC can be programmed to translate Set 2 scancodes, received from the keyboard, to Set 1 scancodes, so that the keyboard device drivers need only know about Set 1 scancodes. Minix 3 keyboard driver configures the KBC to translate Set 2 scancodes, thus, as long as you do not change the KBC configuration in that respect, your code needs to handle only Set 1 scancodes.

3.2 The i8042: The keyboard controller (KBC)

In modern PCs the communication between the keyboard and the processor is mediated by an electronic component that provides the functionality of the i8042, the keyboard controller (KBC). The communication between the KBC and the keyboard (actually, a microprocessor embedded in the keyboard) is by means of a serial communication protocol that is not the object of this lab.

In this lab, you need only to interface with the "i8042", which was thoroughly described in class. In addition to read the class notes, you may find it useful to read the 8042 functional description from the IBM Technical Reference Manual and the data sheet of an 8042-compatible IC. There are several other resources on the Web, including those mentioned in the class notes. You may wish to take a look at these, if you find the material I wrote insufficient.

4. Minix 3 Notes

4.1 Disabling the Default Interrupt Handler

When Minix 3 boots, its terminal driver configures the KBC and installs its own keyboard interrupt handler. This way, users are able to login to Minix and use the different virtual terminals. On one hand this is great: you do not need to configure the KBC. On the other hand this raises an issue: how can your code read the scancodes? Indeed, when the user presses/releases a key, the KBC will generate an interrupt, and the KBC interrupt handler of the terminal driver will execute and read the scancode, and possibly echo a character in the terminal.

A solution to this issue, is for your driver to subscribe the KBC interrupts, as described in Section 5.2 of Lab 3. However, now it should specify not only the `IRQ_REENABLE` policy but also the `IRQ_EXCLUSIVE` policy. As a result, the standard Minix 3 KBC interrupt handler will not be notified of the occurrence of KBC interrupts, thus preventing it from interfering with your code.

Note that the keyboard's IRQ number is 1.

To ensure that you can use Minix 3 virtual terminals once your program is done, your code must cancel its subscription of the KBC interrupt before exiting, by calling the `sys_irqrmpolicy()` kernel call.

4.2 Measuring Time

Both the KBC and the keyboard may take some time to respond to a command. For example, IBM's specification of the i8042 requires the keyboard to respond to a command in 20 ms. Thus your code should not expect to get a response immediately after issuing a command. A simple approach is for your code to wait indefinitely, or until the KBC reports a time-out. A more fault-tolerant approach is for your code to give enough-time for the KBC or the keyboard to respond, retry a few times on time-out, and finally give up. Given that the time intervals to consider are in the order of tens of ms, the `sleep()` function is not useful, because it is able to measures time intervals whose duration is a multiple of a second. Instead, you can use the function `tickdelay()` of Minix 3's `libsys` as follows:

```
#include <minix/sysutil.h>

#define DELAY_US    20000

tickdelay(micros_to_ticks(DELAY_US);
```

This function is similar to `sleep()` in that it blocks the process that executes it for the time interval specified in its argument. After that time interval, the process resumes execution by executing the instruction that follows `tickdelay()`.

4.3 Assembly Programming in Minix

Minix 3 can use the GNU build system. Therefore, you can write your assembly code using the GNU assembler (GAS) assembly language (AT&T's syntax). You should also be able to use the so-called Intel's syntax, which is supported since version 2.10. Indeed, the version of GAS in the Minix version we are using is version 2.17, however I have not tried it. If you wish you can give it a try, but be warned that this syntax appears to be poorly documented (so you may find it useful this link to the GAS section of the Linux Assembly HOWTO).

For an example of a simple assembly function in AT&T's syntax, check pg. 18 of the class notes on Mixed C and Assembly Programming.

4.4 Executing I/O Directly (in Assembly)

Minix 3 device drivers, as your code, although privileged execute at user level. In Minix 3, and in almost all OS's, by default, user level programs are not allowed to execute **I/O sensitive** operations, such as `IN`, `OUT`, `STI` or `CLI`. Therefore, if your interrupt handler issues any of these instructions, it will abort as a result of a general protection exception.

However, the IA-32 ISA provides a mechanism that allows some processes to execute those operations even when running at user-level. This is based on the use of the `IOPL` (I/O Privilege Level) field of the `Extended Flags (EF)` register. This field specifies the minimum current privilege level of a process when it executes those operations. If the current privilege level is lower (meaning that it has more privileges) than that specified in that field of the `EF` register, then the process will be allowed to perform I/O sensitive operations.

Minix 3 offers a kernel call sys_enable_iop() that allows the `IOPL` of a process to be set to a value such that it is able to execute I/O sensitive operations, even though it executes at user level. The following code segment illustrates how this can be done.

```
    sef_startup();

    /* Enable IO-sensitive operations for ourselves */
    sys_iopenable(SELF);
```

5. Testing Code

So that we can grade your work, you are required to implement the test functions described in this section. We will develop the code that will call them, so make sure that your implementation matches their prototypes.

Rather than implement the required functionality directly in these functions, you should design and implement functions that may be useful to interface with the keyboard in your integration project, i.e. functions that read the scancodes sent by the keyboard and that may be of use to configure or to find the status of the keyboard. We will grade not only how you structure the required functionality in functions, but also how do you group these functions in compilation modules, i.e. in the source files.

5.1 `kbd_test_scan(unsigned short asm)`

The purpose of this function is to test that your code is able to read the scancodes from the KBC using an interrupt handler (IH). Depending on whether its argument is zero or not, it should use the IH written in C or the IH written in assembly.

Independently of the implementation language of the IH, it should first subscribe the KBC interrupts, as described in the previous section.

Then, as the IH receives the scancodes from the keyboard, it should print them on the console, indicating whether or not the scancode is a make or a break code. The output should look like the following:

```
Makecode: 0x01
Breakcode: 0x81
```

The `kbd_test_scan()` function should exit when the user releases the `Esc` key, whose break code is `0x81`. For the reasons described in the previous section, it should cancel the subscription of the KBC interrupt before terminating.

IMP. If you do not cancel the subscription of the KBC interrupt, Minix's keyboard interrupt handler will not be notified of interrupts on the keyboard and therefore you will not be able to use any of the virtual terminals on the virtual machine. However, you will be able to use the command line interface (CLI) via a remote shell, for example on a Linux terminal, or in Eclipse. The remote shell uses the keyboard on a system different from Minix.

ASM Implementation

You should use the **linked assembly** approach to mix C and assembly code, i.e. your assembly code and your C code should be in separate files. Furthermore, your ASM implementation of the IH should use the GNU assembler syntax.

IMP. The variables used to pass information between the interrupt handler and the rest of your program should be defined in the assembly file.

IMP. If you use directives of the C pré-processor, e.g. `#include` or `#define`, your file must have the **`.S`** extension, otherwise the C pré-processor will not be invoked (check gcc's documentation).

5.2 `kbd_test_leds(unsigned short n, unsigned short *toggle)`

The purpose of this function is to test your code that changes the state of the keyboard indicator LEDs. Its arguments are an array `toggle` and `n`, the number of elements of `toggle`. Each element of `toggle` specifies which indicator LED should have its state toggled. Valid values for the elements of this array are 0, 1 and 2, denoting the scroll lock, the numeric lock and the caps lock indicators, respectively.

The function `kbd_test_leds()` should process each element of the `toggle` array once per second. For example if `toggle`'s elements are `0,1,1,1,1,0`, your program should turn on the LED corresponding to bit 0, then 1 second later turn on the LED corresponding to bit 1, then 1 second later turn off the same LED, then 1

second later turn on the same LED, then 1 second later, turn it off, and 1 second later turn off the LED corresponding to bit 0.

After processing all elements of the `toggle` array, your function should exit.

To measure the time you can use Timer 0 interrupts, or the `sleep()` function, but in the latter case the maximum possible score will be 90%, rather than 100%.

IMP You should try to implement a layered solution as suggested in the lecture. You will score points for the quality of the design of your functions.

5.3 `kbd_test_timed_scan(unsigned short n)`

The purpose of this function is to test whether your program is able to handle interrupts from more than one device.

Essentially, this test program should be similar to `kbd_test_scan()` in that it should print in the console the scancodes received from the keyboard (here, you can use only the C-version of the IH). The difference is that in this function, your program should exit not only when the user releases the ESC key, but also if it does not receive a scancode for a number seconds equal to its argument, `n`.

To measure this time interval you should use the interrupts of the PC's Timer 0. Note that you need not change the configuration of PC's Timer 0, only subscribe its interrupts, as done in `timer_test_square()` of Lab 2.

6. Compiling your Program

Follow the procedure described for the last lab. In particular, make sure you create a working `Makefile` and that you add it to the directory in your repository for this lab. You will score some points for this.

IMP If you use any assembly file, just add it to the definition of the `SRCS` variable in the Makefile, because **`gcc`** will automatically determine the kind of processing it requires from its extension (check gcc's documentation).

7. Configuring your Program

Your program invokes functions that are privileged. Thus, before you can run it, you need to add a file named `lab3` to the `/etc/system.conf.d/` directory to grant your program the necessary permissions.

Unless you use privileged functions that are not really necessary, the following entry is enough:

```
service lab3
{
        system
                DEVIO
                IRQCTL
                IOPENABLE
                ;
        ipc
                SYSTEM
                rs
                vm
                pm
                vfs
                ;
        io
                40:4
```

```
                60
                64
                ;
        irq
                0       # TIMER 0 IRQ
                1       # KBD IRQ
                ;
        uid     0
                ;
};
```

Once you have created the file with the permissions, you can execute your program using the `service` command. Given that your program should be privileged, you must run it as **user `root`**. Thus, assuming that you are logged in as user `lcom` and your shell's current directory is `/home/lcom/lab3`, you can issue the following commands:

```
$ su
# service run `pwd`/lab3 [-args "< list of space separated arguments >"]
```

Here we assume that your `main()` is similar to the one we provided in lab1.c. (You'll score points for this.)

Note The `` `pwd` `` expression evaluates to the output of the command `pwd`, which returns the present working directory (`/home/lcom/lab3`, according to our assumptions). This is simpler than having to type the full path:

```
# service run /home/lcom/lab3/lab3 [-args "< list of space separated arguments >"]
```

8. Submission

You should submit your code to the SVN repository as you develop and test it. This code should be under directory `lab3`, which should be at the top level of the SVN repository of your Redmine project.

Given that in this lab we give you very few files, we suggest that you use the following SVN commands to setup the repository:

`svn mkdir https://svn.fe.up.pt/repos/lcom1415-t0g00/lab3`
: Note that this need not be given inside a working copy. It can be used to create the directory for this lab directly in the SVN repository. Afterwards, you'll need either to update your working copy if you have one, using the `update` command, or else create a working copy, using the `checkout` command. If you already have a working copy, you may wish to use the following command, which works also for directories.

`svn add <file_name>`
: This "adds" the file named `<file_name>` to the working copy. To be precise, what it does is to **schedule** that file in your working copy to be added to the repository in the next commit. I.e., the file must already exist in a directory that already belongs to the working copy. Furthermore note that, `add` does not add the file to the repository. That will only happen on the next commit of your working copy. As mentioned above, if `<file_name>` is the name of a directory that already exists, it will add the it to the working copy.

These commands are rather flexible. You should read their documentation in Chapter 9 of the SVN 1.6 Red Book. In particular, you should be aware of the `revert` command, just in case you change your mind before commiting.

The deadline for the submission is at the end of your second class of this lab. Because, some sections cannot stay in the lab after the end of their classes, you have only a 10 minutes tolerance. That is we will grade the latest submission as of 10 minutes after the end of your 2nd class. Please make sure that you are sufficiently familiar with SVN to avoid last minute problems.

IMP. You should add to the SVN repository all files needed so that compilation of your code can be done by checking out the `lab3` directory, and executing:

```
make
```

under that directory in the working copy. You'll score points for this.

Acknowledgments

This lab is based on a lab by João Cardoso and Miguel P. Monteiro for DJGPP running on Windows98.