**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 via interrupts;
2. Read and display the scancodes, both the make codes and the break codes, generated by the PC's keyboard via polling;

Furthermore, we want you to learn:

1. How to handle interrupts from more than one I/O device

Unlike in Lab 2 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()
2. int kbd\_test\_poll()
3. int kbd\_test\_timed\_scan(uint8\_t idle)

These functions are declared in header file lab3.h, and source file [lab3.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c) contains their implementation stubs. You should add your test code to that file, because it is already prepared for 1) use with the LCF, and 2) incremental development. [Section 5](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/lab3_05.html) describes what these functions should do.

Because lab3.h is indirectly included via <lcom/lcf.h>, we do not provide any link to it. Instead, we provide you the relevant [doxygen documentation](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/doc/index.html), which is all you need for implementing the functions specified here.

**2.1 Class Preparation**

This lab is planned for two lab classes.

**First class**

The goal for the first class is to implement:

1. int kbd\_test\_scan(), with the IH written in C
2. int kbd\_test\_poll()

Therefore, for the first class you should read the material presented in the first lecture 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. Copy file [lab3.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c) to the folder /home/lcom/labs/lab3, that already exists (in Minix's filesystem):
2. Import the directory tree rooted at lab3 to the SVN repository of your project on Redmine, as described in [Lab 0 's Section 6.1](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab0/lab0_6.html#importing). (Remember that i) this need not be done in a working copy, and ii) it does not make the imported directory a working copy.)
3. Transform ~/labs/lab3/ in a working copy of the directory tree rooted at lab3, as described in [Lab 0 's Section 6.2](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab0/lab0_6.html#working_copy).

**Second class**

The goal for the second class is to implement:

1. int kbd\_test\_timed\_scan(uint8\_t idle)

Therefore, for the second class you should read also Sections 4.3, 4.4 and 5.3 of this handout.

### 3. The PC Keyboard and its Controller

Whenever a user presses or releases one of its keys, the keyboard sends a **scancode** to the keyboard controller (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 is seldom used in polled mode, you will also read the scan codes using polling in this lab.

#### 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) of the characters that labels it. 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 Portuguese keymap. The advantage of this approach is clear for keyboard manufacturers: a key in a given position, always generates the same scancode, 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 set of scancodes that we have described are known as **Set 1**, that is 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 know only 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](https://web.fe.up.pt/~pfs/aulas/lcom2019/at/7kbd.pdf), you may find it useful to read the [8042 functional description from the IBM Technical Reference Manual](http://zet.aluzina.org/images/d/d4/8042.pdf) and [the data sheet of an 8042-compatible IC](http://www.alldatasheet.com/datasheet-pdf/pdf/144616/WINBOND/W83C42.html). 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 that 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 5.3 of Lab 2](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab2/lab2_05.html#int). 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.

**IMP.:** 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 its resolution is one second, i.e. it is able to measure 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().

### 5. Test 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. You should try to implement a layered solution as suggested in the lectures. You will score points for the quality of the design of your functions. 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.

**IMP.-**Your implementation must use the LCF. Therefore, all your C source code files must include the following line:

#include <lcom/lcf.h>

This should be the first header file to be included in all your C source code files. Furthermore, any other header files that you include explicitly, i.e. via the #include directive, must have been created by you. .

Note that we do not provide any link to lab3.h, because it is already included via <lcom/lcf.h> we are not providing any link to lab3.h. Instead, we provide you the relevant [doxygen documentation](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/doc/index.html), which is all you need for implementing the functions specified here.

#### 5.1 kbd\_test\_scan()

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) written in C.

In its definition, this function's name must appear between parenthesis, as shown in [lab3.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c):

#include <lcom/lcf.h>

int (kbd\_test\_scan)() {

/\* To be completed \*/

}

We suggest that you write it by adding your code to the stub provided in [that file](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c).

Independently of the implementation language of the IH, kbd\_test\_scan() should first subscribe the KBC interrupts, as described in the previous section.

The IH must be a function that takes no arguments and returns no value. The passing of data between the IH and the other functions of your implementation must be done via global variables. The name of the IH must be

#include <lcom/lcf.h>

void kbc\_ih();

The handler should read the status from the status register, to check if there was some communications error, and to read the scancode byte from the output buffer.

In its definition, the function name of the kbc\_ih() function must appear between parenthesis, as shown:

#include <lcom/lcf.h>

void (kbc\_ih)() {

Please note that lab3.h includes the prototype of the interrupt handler (IH), because the LCF needs to intercept calls to the IH. However, this does not mean that you should implement it in lab3.c, because that would break the layers that you are supposed to use to organize your code. Please note that this has no implication whatsoever in the header files that you must include whenever you want to invoke these functions: as we have described, lab3.h is always included via <lcom/lcf.h>, which you must include in every C source file.

Every time the IH receives a byte from the keyboard, kbd\_test\_scan() should try to assemble a scancode. If it succeeds it should print it on the console, by calling the following function that **we provide** you:

#include <lcom/lcf.h>

int kbd\_print\_scancode(bool make, uint8\_t size, uint8\_t bytes[]);

where:

**make**

indicates whether this is a make or a break code

**size**

is the size of the scancode in bytes

**bytes**

is an array with the bytes of the scancode, in order, i.e. the first byte will be in the first element of the array, the second byte, if any, in the array's second element and so on

The output of this function will look like the following:

Makecode: 0x12

Breakcode: 0x92

Makecode: 0xeO 0x49

Breakcode: 0xe0 0xc9

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. Also, before returning, this function should print the number of sys\_inb kernel calls by calling the following function that **we provide** you:

#include <lcom/lcf.h>

int kbd\_print\_no\_sysinb(uint32\_t cnt);

where:

**cnt**

is the total count of sys\_inb() kernel calls by your program.

Because of the LCF, to count the number of sys\_inb() calls, you must use the approach based on a wrapper function, as described in [the lecture notes](https://web.fe.up.pt/~pfs/aulas/lcom2019/at/7kbd.pdf).

**IMP.-** Header file lab3.h includes also the prototypes of the functions that we are providing you and that you will have to call from your code: kbd\_print\_scancode() and kbd\_print\_no\_sysinb(). Thus, you need to include <lcom/lcf.h> in all C files where you call these functions (but you should have done that already).

#### 5.2 kbd\_test\_poll()

The purpose of this function is to test that your code is able to read the scancodes from the KBC using polling, i.e. you cannot use interrupts.

In its definition, this function's name must appear between parenthesis, as shown in [lab3.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c):

#include <lcom/lcf.h>

int (kbd\_test\_poll)() {

/\* To be completed \*/

}

We suggest that you write it by adding your code to the stub provided in [that file](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c).

Like kbd\_test\_scan(), kbd\_test\_poll() should read the scan codes sent by the keyboard and print them on the console, by calling function kbd\_print\_scancode(), already described in the previous section. Therefore, the output will look like the following:

Makecode: 0x01

Breakcode: 0x81

Likewise, kbd\_test\_poll() should exit when the user releases the Esc key, whose break code is 0x81, and print the number of sys\_inb() kernel calls, by calling the function kbd\_print\_no\_sysinb() also described in the previous section.

**Food for thought** Can you explain the large difference between the numbers of sys\_inb() kernel calls displayed by kbd\_test\_poll() and by kbd\_test\_scan()?

**IMP.-** Note that before exiting, kbd\_test\_poll() must enable interrupts, by writing an appropriate KBC command byte. (You will get marks for that.) If it does not do it, the KBC will not generate interrupts, and thus the keyboard will get stuck. This is because lcf\_start() disables keyboard interrupts by writing an appropriate command byte to the KBC, before calling kbd\_test\_poll(), to prevent Minix's keyboard IH from stealing the scancodes from your program,.

To make your life easier, lcf\_start() resets the KBC to a proper state, even if kbd\_test\_poll() does not enable interrupts, as described in the previous paragraph. However, the keyboard will remain stuck until lcf\_start() is invoked again. Therefore, you will have to run your program again from a remote shell.

#### 5.3 kbd\_test\_timed\_scan(uint8\_t idle)

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

In its definition, this function's name must appear between parenthesis, as shown in [lab3.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c):

#include <lcom/lcf.h>

int (kbd\_test\_timed\_scan)(uint8\_t n) {

/\* To be completed \*/

}

We suggest that you write it by adding your code to the stub provided in [that file](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/src/lab3.c).

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, by invoking kbd\_print\_scancode() 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 of seconds equal to its argument, idle.

To measure this time interval you **must** 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\_int() of [Lab 2](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab2/lab2.html).