**2. What to Do**

Write in the C programming language several functions that use the PS/2's mouse low level interface. The key functionality to implement is:

1. Read and display the packets sent by the PS/2's mouse
2. To handle interrupts from both the mouse and the timer
3. To use the PS/2 mouse in remote mode
4. Recognize a simple "mouse gesture" using a state machine

Like in Lab 3 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 testing functions:

1. int mouse\_test\_packet()
2. int mouse\_test\_async()
3. int mouse\_test\_remote()
4. int mouse\_test\_gesture()

These functions are declared in header file lab4.h, and source file [lab4.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/lab4.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/lab4/lab4_5.html) describes what these functions should do.

Because lab4.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/lab4/src/doc/index.html), which is all you need for implementing the functions specified [Section 5](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4_5.html).

**2.1 Class Preparation**

So that you can accomplish this lab's objectives, you should do some homework, i.e. some work before the lab class. In addition to read, and understand, this handout and the class notes, you should:

1. Copy file [lab4.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/lab4.c) to the folder /home/lcom/labs/lab4, that already exists (in Minix's filesystem). You should also copy to this folder a makefile you have used in one of the previous labs. (Do not forget to make appropriate changes to this makefile so that it can be used for this lab.)
2. Import the directory tree rooted at ~/labs/lab4 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).
3. Transform ~/labs/lab4/ in a working copy of the directory tree rooted at lab4, as described in [Lab 0's Section 6.2](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab0/lab0_6.html#working_copy).
4. Write mouse\_test\_packet()
5. Write mouse\_test\_remote()

**3. The PS/2 Mouse and the i8042**

The PS/2 mouse has typically 3 buttons and is able to track the movement of the mouse in a plane. Usually, it is configured to report its state, i.e. the state of the buttons or its position in the plane, to its controller in an event-driven fashion, i.e. whenever its state changes. It does so by sending a multi-byte packet to its controller, which puts each of the bytes received in a register, and if configured to generate interrupts, will do so. On the PC, the mouse controller raises IRQ line 12. It is then up to the interrupt handler to read the bytes of the packet, one per interrupt, from the controller.

**3.1 Mouse Packets**

The packets sent by a mouse to its controller are composed by several bytes. Different mouse types use different packet types. For example, whereas the PS/2 mouse uses a 3-byte packet, the Microsoft Intellimouse, which includes a scrolling-wheel, uses a 4-byte packet mouse.

The beauty of the simple interface between the mouse and its controller is that it is rather flexible and can be used to support a wide range of mice, from very simple to very sophisticated. In this Lab, you'll need only use the standard PS/2 protocol, which was presented in the lectures. Any of the [references](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4.html#references) of this handout has a good description of the format of the PS/2 mouse packets.

**3.2 The i8042: The keyboard (and mouse) controller (KBC)**

In modern PCs, the communication between the mouse and the processor is mediated by an electronic component that provides the functionality of the i8042, the keyboard controller (KBC). I.e., the KBC interfaces with both the AT-keyboard and the PS/2-mouse. The communication between the KBC and the mouse (actually, a microprocessor embedded in the mouse) is by means of a serial communication protocol similar to that used in the communication between the KBC and the keyboard, and is not the object of this lab.

In this lab, you need only to interface with the "i8042". In PS/2 mode, this controller supports some mouse related commands, such as enabling/disabling the mouse interface or enabling/disabling interrupt generation upon reception of a byte from the mouse. As usual, these "KBC-commands" are written to port 0x64. Furthermore, it allows the device driver to issue commands directly to the mouse, by using the KBC-command 0xD4, Write (byte) to the Auxiliary Device.

The commands for the mouse, and their arguments, if any, are arguments of the 0xD4 KBC-command. That is, to issue a command to the mouse, the driver must first write command 0xD4 to the KBC, i.e. using port 0x64, and afterwards the mouse command to port 0x60. Like the keyboard, the mouse will send an acknowledgment (message), which indicates whether or not the command was successfully received. The KBC puts this reply in the output buffer, and it must be read from port 0x60. If the command has any arguments, they should be written using the same protocol. I.e., for each byte of the arguments, the driver must first write command 0xD4 to port 0x64, and afterwards the byte to port 0x60. As before, the mouse will send an acknowledgment to the byte sent by the KBC, and the KBC will put it in the output buffer. Again, each acknowledgment must be read from port 0x60. Finally, after receiving the last byte of a command (either the command itself, or the last byte of its arguments), the mouse will execute the command, and if it elicits a response, the mouse will send it to the KBC, which again will put it in the output buffer, and must be read from port 0x60.

Summarizing, for each byte sent using command 0xD4, either a command or an argument, the mouse will send back an acknowledgment, which is put in the output buffer, and must be read from port 0x60. Furthermore, if the mouse command elicits a response, the mouse will send it after the acknowledgment to the last byte of the command (either the command itself or the last byte of its arguments), it will be put in the output buffer, and must be read from port 0x60.

**3.3 Synchronization Issues**

A PS/2-mouse packet is a sequence of 3 bytes. The device driver must be synchronized with the mouse to ensure that when it processes a packet, the 3 bytes it uses all belong to the same packet. For example, using two bytes of a packet and the first byte of the following packet may lead to incorrect behavior.

While testing a solution to this lab, I have found that usually my "driver" was out of sync with the mouse in the very beginning. For example, even though I did not move the mouse, it would report overflow when I pressed a button. This, happened, even if I reseted the mouse. I suspect that this is caused by interference by the Minix 3 KBC code, but I have not investigated the issue further. The important is that, if the driver is out of sync, for example, because one of the bytes is corrupted, it must synchronize again.

However, the bytes of the PS/2 packet do not carry an identification, and therefore it is not easy to detect that the code is not in sync. The solution I found relies on the fact that bit 3 of byte 1, must be 1. Thus, if the byte your code is expecting is the first one, and bit 3 of the byte received is 0, the byte received cannot be the first one and the code is not in sync with the mouse.

Although this does not guarantee that your code will always be in sync, other bytes can have bit 3 set to 1, I've found that this would solve the problem. Actually, the efficacy of this approach will depend on the mouse usage pattern: if the user starts only by clicking the mouse, then bit 3 of all packets but the first will be 0, and in 1 or 2 packets, the "driver" will get in sync with the mouse.

**3.4 Other Remarks**

Finally, I'd like to call your attention to two paragraphs in the [Synaptics Interfacing Guide](https://www.aquaphoenix.com/hardware/ledlamp/reference/synaptics_touchpad_interfacing_guide.pdf). The first one refers to some steps that must be taken before issuing any command to the mouse, to prevent interference from packets sent by the mouse.

*"If the device is in Stream mode (the default) and has been enabled with an Enable (0xF4) command, then the host should disable the device with a Disable (0xF5) command before sending any other command." Synaptics TouchPad Interfacing Guide, pg. 33*

The second one concerns the actions that should be taken upon the reception of a negative acknowledgment to some byte written using the 0xD4 KBC command.

*“When the host gets an 0xFE response, it should retry the offending command. If an argument byte elicits an 0xFE response, the host should retransmit the entire command, not just the argument byte.” Synaptics TouhcPad Interfacing Guide, pg. 31*

**3.5 Other Resources**

You can find an overview of both the mouse-related KBC commands and the mouse commands in [the lecture notes](https://web.fe.up.pt/~pfs/aulas/lcom2019/at/10mouse.pdf). A detailed description of the mouse commands can be found in [Synaptics Interfacing Guide](https://www.aquaphoenix.com/hardware/ledlamp/reference/synaptics_touchpad_interfacing_guide.pdf). You may also find interesting [the data sheet of an 8042-compatible IC that supports the PS/2 mouse](http://www.alldatasheet.com/datasheet-pdf/pdf/113936/WINBOND/W83C43.html). There are several other resources on the Web, including the other two mentioned in the lecture notes, which include some information regarding protocols other than the PS/2.

### 4. Minix 3 Notes

#### 4.1 Minix's Initial Mouse Configuration

Although Minix 3 boots up in command line mode and does not use the mouse, its terminal driver configures the mouse in **stream mode** and installs its own mouse interrupt handler (IH), the default one. Furthermore, to prevent the mouse from generating useless interrupts, Minix disables (stream mode) data reporting.

This Minix configuration of the mouse at boot up means that, so that you can use the mouse and the Minix's default mouse IH does not interfere with your program, either the LCF or your code may have to perform some actions:

enabling data reporting

so that the mouse sends packets reporting its motion or changes in the state of its buttons

setting remote mode

so that the mouse sends packets only upon request

mouse interrupts subscription

to prevent Minix's default IH from "stealing" mouse packets. (This requires using not only the IRQ\_REENABLE policy but also the IRQ\_EXCLUSIVE policy.)

Which actions have to performed by your program and which actions must be performed by the LCF, depends on the test function and is specified in [Section 5](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4_05.html). Regardless, your program shall revert them before exiting. This will ensure that the configuration of the mouse upon exit of your program, will be the one when it started.

#### 4.2 Waiting for the Mouse to Respond

Like the keyboard, the mouse communicates with the KBC via a serial line, and thus you should not expect to receive the acknowledgment to a byte you write to the mouse, or the response it sends to a command, immediately after issuing a command. As suggested for Lab 3 rather than wait indefinitely, or until the KBC reports a time-out, your code should give enough-time for the KBC or the mouse 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, it is not appropriate to use sleep(), which measures time intervals whose duration is a multiple of a second. Instead, you can use the function tickdelay() of Minix 3's libsys as described in [Section 4.2 of Lab 3](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab3/lab3_04.html#sec4.2)

### 5. Test Functions Specification

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 mouse in your integration project, i.e. functions that read the packets sent by the mouse and that may be of use to configure or to find the status of the mouse. 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.

#### 5.1 LCF Requirements

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. You may create your own header files and use the #include directive to include them. However, note that you **must not** declare any of the functions that are [specified for this lab](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/index.html) in your own header files: they are already declared in header files included via <lcom/lcf.h>.

Another requirement of the LCF is that, in the definition of each of the 4 functions specified in this Section, the name of the function must appear between parenthesis. For example, for the first function, you should do as done in the respective stub in [lab4.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/lab4.c):

#include <lcom/lcf.h>

int (mouse\_test\_packet)(uint32\_t cnt) {

/\* To be completed \*/

printf("%s(%u): under construction\n", \_\_func\_\_, cnt);

return 1;

}

Because [lab4.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/lab4.c) already has a stub for each of the main functions that you have to implement and the name of the function of each of these stubs is already between parentheses, we suggest that you write these functions by editing [lab4.c](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/lab4.c).

#### 5.2 int mouse\_test\_packet(uint32\_t cnt)

The purpose of this function is that you learn 1) how to read, using an interrupt handler (IH), the packets the PS/2 mouse sends when operating in stream mode; and 2) how to parse the mouse packets. Therefore, this function must configure the mouse properly, receive the number of mouse packets specified in its argument via interrupts, parse them and print them in a friendly way. Before exiting, it shoud reset the mouse to Minix's default configuration. You can find its documentation [here.](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__lab4.html)

As explained in [Section 4](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4_4.html), because of Minix's default mouse configuration, mouse interrupts must be subscribed in exclusive mode (remember that the mouse raises IRQ 12), and data reporting by the mouse must be enabled. To help you getting started, we are providing, via the LCF library, the int mouse\_enable\_data\_reporting() function:

#include <lcom/lcf.h>

int mouse\_enable\_data\_reporting();

which enables data reporting in stream mode, as [documented here](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__mouse.html). We expect that, in your final version, i.e. the one available in the SVN repository by the submission deadline, data reporting in stream mode will be enabled by your own code, rather than by calling the function we are providing you. You will score points for that.

As mentioned in the lectures, upon each interrupt, the IH must read only one byte. It's prototype must be as [documented here](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__mouse.html):

#include <lcom/lcf.h>

void mouse\_ih();

Since the IH takes no arguments and returns no values, the passing of data between mouse\_ih() and the other functions of your implementation must be done via global variables, preferably static.

Remember that, in its definition, the function name of the mouse\_ih() function must also appear between parenthesis, as shown:

#include <lcom/lcf.h>

void (mouse\_ih)() {

You should implement this function in a source file together with related functions, as discussed in the lectures.

Every time a packet is received, i.e. upon receiving the 3rd byte of a mouse packet, your program should parse it and print it on the console, by calling the following function that is [documented here](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__test4.html) and that we **provide** you as part of the LCF:

#include <lcom/lcf.h>

void mouse\_print\_packet(struct packet \*pp);

where:

**pp**

is the address of a struct packet, which is [documented here](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/structpacket.html)

[Figure 1](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4_5.html#figure1) illustrates the output generated by mouse\_print\_packet().

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| B1=0x08 | B2=0x12 | B3=0x14 | LB=0 | MB=0 | RB=0 | XOV=0 | YOV=0 | X=18 | Y=20 |
| B1=0x08 | B2=0x12 | B3=0x12 | LB=0 | MB=0 | RB=0 | XOV=0 | YOV=0 | X=18 | Y=18 |
| B1=0x08 | B2=0x12 | B3=0x0e | LB=0 | MB=0 | RB=0 | XOV=0 | YOV=0 | X=18 | Y=14 |
| B1=0x08 | B2=0x10 | B3=0x0e | LB=0 | MB=0 | RB=0 | XOV=0 | YOV=0 | X=16 | Y=14 |
| B1=0x09 | B2=0x00 | B3=0x00 | LB=1 | MB=0 | RB=0 | XOV=0 | YOV=0 | X=0 | Y=0 |
| B1=0x0C | B2=0x00 | B3=0x00 | LB=0 | MB=1 | RB=0 | XOV=0 | YOV=0 | X=0 | Y=0 |
| B1=0x0A | B2=0x00 | B3=0x00 | LB=0 | MB=0 | RB=1 | XOV=0 | YOV=0 | X=0 | Y=0 |

**Figure 1**: Example of output generated by mouse\_print\_packet()

Before calling mouse\_print\_packet(), your code must initialize the struct packet whose address is pp. Because the struct packet contains not only the bytes of a mouse packet, but also members such as the displacement in both the x and the y directions, or the state of the mouse buttons, your code must parse the packets received from the mouse.

The mouse\_test\_packet() function should terminate after printing the number of packets specified in its argument **cnt**. However, before returning, it should reset the mouse to its state before the execution of your program, i.e. it must disable data reporting and unsubscribe mouse interrupts.

#### 5.3 mouse\_test\_remote(uint16\_t period, uint8\_t cnt)

The purpose of this function, which is [documented here](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__lab4.html), is that you learn how to read the packets from the PS/2 mouse using remote mode and by polling the KBC's status register, rather than using streaming mode and interrupts, as in mouse\_test\_packet().

mouse\_test\_remote() should configure the mouse to operate in remote mode and periodically request a data packet from the mouse. The period to be used is specified in milliseconds in its first argument. Reading of mouse packet bytes should be performed by polling the status register. Like mouse\_test\_packet(), mouse\_test\_remote() should assemble, parse and print in a human friendly format, by calling mouse\_print\_packet(), the mouse packets. mouse\_test\_remote() should return after it prints the number of packets specified in its argument **cnt**.

To make your life easier, lcf\_start() configures both the mouse and the KBC as appropriate: it configures the mouse to operate in remote mode and disables mouse interrupts by issuing the appropriate KBC command byte. This ensures that upon return from lcf\_start(), the mouse and the KBC are ready to respond to read data commands as required.

mouse\_test\_remote() should reset both the KBC and the mouse to an appropriate state before returning. That is, it must reset the mouse to stream mode with data reporting disabled. Furthermore, it should reset the KBC command byte to Minix's default value, which should be obtained by calling the minix\_get\_dflt\_kbc\_cmd\_byte() function:

#include <lcom/lcf.h>

uint8\_t minix\_get\_dflt\_kbc\_cmd\_byte();

minix\_get\_dflt\_kbc\_cmd\_byte() is provided to you as part of the LCF and is [documented here](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__minix.html).

#### 5.4 mouse\_test\_async(uint8\_t idle\_time)

The purpose of this function is to make you think about the structure of the code that handles asynchronous interrupt notifications from multiple devices, namely the PC's timer 0 and the mouse.

This function should do essentially the same as mouse\_test\_packet(), i.e. it should display the packets received from the mouse, as shown in [Figure 1](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4_5.html#figure1).

The difference is on the exit condition: mouse\_test\_async()should terminate if it receives no packets from the mouse for the number of seconds specified in its argument **idle\_time**.

For measuring the time you must use timer 0's interrupts. Furthermore, you must not change the configuration of timer 0. You can determine timer 0 interrupt frequency with the help of the kernel call sys\_hz().

Like in mouse\_test\_packet(), mouse\_test\_async() must subscribe mouse interrupts and enable data reporting, for which it can use [mouse\_enable\_data\_reporting()](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__mouse.html), until you develop that functionality. Also like in mouse\_test\_packet(), it is up to mouse\_test\_async() to disable data reporting and unsubscribe mouse interrupts before returning.

#### 5.5 mouse\_test\_gesture(uint8\_t x\_len, uint8\_t tolerance)

The purpose of this function is that you learn how to implement a state machine, whose states change in response to events originating from I/O devices, more specifically the mouse.

This function should do essentially the same as mouse\_test\_packet(), i.e. it should display the packets received from the mouse, as shown in [Figure 1](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/lab4_5.html#figure1).

The difference is on the exit condition: now, the function should terminate if the user draws a **logical and symbol**, i.e. an inverted **V**, from left to write. The first line must be drawn while the left button, and no other, is pressed down, whereas the second line must be drawn while the right button, and no other, is pressed down. The absolute value of the slope of each line must be larger than 1 and the value of the displacement of each line along the x-direction must have the minimum value specified in the first argument, x\_len. The units of both arguments are the units reported by the mouse.

Your gesture matching algorithm must also satisfy the following requirements:

1. The start of a line is marked by the pressing down of the relevant button when all buttons are released.
2. Each line must be drawn in the "right" direction. To tolerate less precise mice, you shall give the tolerance specified in the second, tolerance, argument. E.g. the first line should be upwards, therefore all movements reported by the mouse should, in principle, have non-negative displacements along both the x and the y directions. But your algorithm must allow "small" negative displacements along both the x and the y directions, i.e. their absolute value cannot be larger than the second argument, tolerance.
3. The end of a line is marked by the release of the relevant button.
4. On the vertex, the left button must be released before the right button is pressed down. No other button-related events are acceptable between these two.
5. Every movement reported between the two button-related events that define the vertex must be residual only, i.e. the absolute value in both the x and the y directions cannot be larger than the value of the second, tolerance, argument.
6. To simplify the state machine, especially its hierarchical/layered implementation, you must ignore any displacement reported on a packet that generates either the start-of-line or the end-of-line button-based events. Essentially this means that each mouse packet can lead to at most one event relevant for gesture matching.

The algorithm must start matching the gesture from the very beginning, as soon as it determines that the mouse events do not satisfy the requirements in the previous list (or the requirements specified in the paragraph that precedes this list).

Like in mouse\_test\_packet() (and mouse\_test\_async()), mouse\_test\_gesture() must subscribe mouse interrupts and enable data reporting, for which it can use [mouse\_enable\_data\_reporting()](https://web.fe.up.pt/~pfs/aulas/lcom2019/labs/lab4/src/doc/group__mouse.html), until you develop that functionality. Also like in mouse\_test\_packet() (and mouse\_test\_async()), it is up to mouse\_test\_gesture() to disable data reporting and unsubscribe mouse interrupts before returning.