Computer Labs 2014/2015 - 1st Semester
Lab 4: The PS/2 Mouse

---

1. Objectives

The objectives of this lab are threefold:

1. that you learn the operation of the PS/2 mouse and how to use its low level interface.
2. that you get a better understanding of the interrupt mechanism and that you learn how to use it in the context of the PS/2 mouse
3. that you learn how to implement state machines in event-driven designs

This lab is planned for 1 lab class, but you are expected to work also during Semana da FEUP. More specifically, in addition to attend the lectures and spend 5 hours preparing for the lab class, you are expected to work about 8 hours during Semana da FEUP.

This handout is structured as follows:

2. What to Do

Write in C 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. Process the interrupts generated asynchronously by more than one device
3. To display the configuration of the mouse
4. Recognize a simple "mouse gesture" using a state machine

With respect to the first task, the communication between the keyboard controller (KBC) interrupt handler and the part of your program that displays the packets should use a simple array whose elements are of type `unsigned char` and a counter variable of type `unsigned short` that keeps track of the bytes received.

Like in Lab 4 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 test_packet()
2. int test_async()
3. int test_config()
4. int test_gesture()
```

These functions are declared in header file `test4.h`, and file `test4.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.

IMP. In addition to these files, you should submit the file `lab4.c`, which should contain only the `main()` function, and will be used to grade the flexibility of your test code. For testing your code, we will use our own `main()` function.

2.1 Class Preparation

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

1. Create directory/folder `lab4` at the top level of your SVN repository, and add all files for this lab, under this folder
2. Write `test_packet()`
3. Write `test_async()`

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. 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 types of mouse use different types of packets. 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 class. Any of the references at the end of this document 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. 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 class notes. A detailed description of the mouse commands can be found in Synaptics Interfacing Guide. You may

also find interesting the data sheet of an 8042-compatible IC that supports the PS/2 mouse. There are several other resources on the Web, including the other two mentioned in the class notes, which include some information regarding protocols other than the PS/2.

4. Minix 3 Notes

4.1 Disabling the Default Interrupt Handler

Although Minix 3 boots up in command line mode and does not use the mouse, its terminal driver configures the mouse and installs its own mouse interrupt handler. So, you need not initialize the mouse, and can use Minix 3's configuration. However, to prevent this handler from stealing the mouse packets away from your program, you should do as described in Section 4.1 of Lab 3. I.e., when subscribing the mouse interrupts, your program should specify not only the `IRQ_REENABLE` policy but also the `IRQ_EXCLUSIVE` policy. As a result, the standard Minix 3 mouse interrupt handler will not be notified of the occurrence of mouse interrupts, thus preventing it from interfering with your code.

Although Minix 3 in command line mode does not use the mouse, your code should cancel its subscription of the mouse interrupt before exiting, by calling the `sys_irqrmpolicy()` kernel call

Note that the mouse's IRQ number is 12.

4.2 Measuring Time

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

5. Testing Code

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

When designing your solution always think about modularity and generality. The grade of your code will depend also on these aspects.

5.1 `test_packet(unsigned short cnt)`

The purpose of this function is to test that your code is able to read the packets from the PS/2 mouse using an interrupt handler.

Thus, `test_packet()` should first subscribe the mouse interrupts, as described in the previous section.

Then it should print in a user friedly way on the console the packets received by the interrupt handler, as shown in Figure 1.

```
B1=0x8  B2=0x12  B3=0x14  LB=0  MB=0  RB=0  XOV=0  YOV=0  X=18  Y=20

B1=0x8  B2=0x12  B3=0x12  LB=0  MB=0  RB=0  XOV=0  YOV=0  X=18  Y=18

B1=0x8  B2=0x12  B3=0xe   LB=0  MB=0  RB=0  XOV=0  YOV=0  X=18  Y=14

B1=0x8  B2=0x10  B3=0xe   LB=0  MB=0  RB=0  XOV=0  YOV=0  X=16  Y=14

B1=0x9  B2=0x0   B3=0x0   LB=1  MB=0  RB=0  XOV=0  YOV=0  X=0   Y=0
```

```
B1=0xC  B2=0x0   B3=0x0   LB=0  MB=1  RB=0  XOV=0  YOV=0  X=0   Y=0

B1=0xA  B2=0x0   B3=0x0   LB=0  MB=0  RB=1  XOV=0  YOV=0  X=0   Y=0
```

Figure 1: Example of output generated by `test_packet`

The `test_packet()` function should exit after it prints the number of packets specified in its argument **`cnt`**.

5.2 `test_async(unsigned short 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 essentially do the same as `test_packet()`, i.e. it should display the packets received from the mouse, as shown in Figure 1.

The difference is on the exit condition: now, the function should terminate 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 the Timer 0 interrupts. Furthermore, you cannot change the configuration of Timer 0.

5.3 `test_config()`

The purpose of this function is to test your code that displays the configuration of the mouse. The mouse configuration should be presented in the console in a user friendly way. Your program should exit after displaying the mouse configuration.

5.4 `test_gesture(short length, unsigned short tolerance)`

The purpose of this function is to test whether you are able to implement a state machine, whose states change in response to events originating in I/O devices, more specifically the mouse.

This function should essentially do the same as `test_packet()`, i.e. it should display the packets received from the mouse, as shown in Figure 1.

The difference is on the exit condition: now, the function should terminate if the user makes a continuous "horizontal" movement in the same direction with the mouse **while pressing its left key**. The minimum length of the movement is specified in the `length` argument. The units of this argument are the units reported by the mouse. A positive number indicates a movement towards the right, whereas a negative number indicates a movement in the other direction.

The second argument specifies the vertical movement, in either direction, that is tolerated to consider the movement as horizontal. Again, the units are those used by the mouse.

Hint To simplify the implementation, reset movement detection whenever restrictions are violated, e.g. the vertical tolerance is exceeded.

6. Compiling your Program

Follow the procedures described for the previous labs (of course now you should use directory `lab4` instead of directory `lab3` or `lab2`), but do not forget to modify the `Makefile`.

7. Configuring your Program

Your program invokes functions that are privileged. Thus, before you can run it, you need to add a file named `lab4` 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 lab4
{
        system
                DEVIO
                IRQCTL
                ;
        ipc
                SYSTEM
                rs
                vm
                pm
                vfs
                ;
        io
                40:4
                60
                64
                ;
        irq
                0         # TIMER 0 IRQ
                1         # KBD IRQ
                12        # AUX/MOUSE IRQ
                ;
        uid     0
                ;
};
```

I suggest that you add the new entries (actually only the IRQ for the mouse) to the configuration file you have used in previous labs.

IMP. Do not forget to change the name of the service in the first line to `lab4`.

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/lab4`, you can issue the following commands:

```
$ su
# service run `pwd`/lab4 [-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/lab4`, according to our assumptions). This is simpler than having to type the full path:

```
# service run /home/lcom/lab4/lab4 [-args "< list of space separated arguments >"]
```

8. Submission

The **deadline** for the submission of your work is **4 hours after the end of your lab class.**

There is no tolerance, beyond this one. Thus, as you complete the different milestones of this lab, you should commit your work to the SVN repository in Redmine.

I suggest that you also commit your code even if you have not completed the new functionality you are working on. I.e., I suggest that you use the SVN repository also as a backup. Although this is not the proper way of using a SVN repository, it may save your day. (As the platform we have been using appears not to be robust enough for some of you.)

If you wish us to grade a version that is different from the latest one by the deadline, please add a file with name **rev<rev_num>** under folder `lab4` of your repository, where **<rev_num>** is the revision number you wish us to grade. In this case, please be careful not to make any mistake, as it will make us grade the wrong version of your work. Note that this file should be added to the repository by the deadline, otherwise we will not be aware it exists.

IMP. You should add to the SVN repository all files needed so that compilation of your code can be done by checking out the `lab4` directory, and executing:

```
make
```

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

References

- The PS/2 Mouse class notes
- Synaptics Interfacing Guide
- Andries Brouwer's The PS/2 Mouse, Ch. 13 of Keyboard scancodes
- Adam Chapweske's The PS/2 Mouse Interface
- Data sheet of W83C43 an 8042-compatible IC that supports the PS/2 mouse

Acknowledgments

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