USB Keylogger

Minimal USB driver implementation to work with a keyboard

Autor: Arnau Bagó Castro
Director: Juan José Costa
Co-Director: Alberto Miranda

SEPTEMBER 2016
Abstract

This report presents the study and development of the project *USB keylogger, minimal USB driver implementation to work with a keyboard*. This project purpose is to help future computer engineers or computer enthusiasts understand how a USB driver works and how to create a custom USB driver for specific purposes. The final result will be a document, stating the different protocols used in this kind of drivers, and an implemented example, which will be a custom USB driver able to communicate a keyboard with a specific operating system.
Index

1 Glossary ............................................... 1

2 Context and state-of-the-art .................. 3
   2.1 Context .......................................... 3
      2.1.1 Relevance and motivation .............. 3
      2.1.2 Objectives and problem formulation ... 4
      2.1.3 Actors ...................................... 6
   2.2 State-of-the-art ............................. 7
      2.2.1 Technology used ......................... 7
      2.2.2 Similar projects ......................... 7

3 Scope of the project and methodology .... 9
   3.1 Project boundaries ......................... 9
   3.2 Methodology .................................. 10
   3.3 Tools .......................................... 10
   3.4 Validation methodology ................... 11

4 Project planning ................................. 12
   4.1 Project duration ............................ 12
   4.2 Resources .................................... 13
   4.3 Gantt diagram and tasks description ... 13
7.3.1 Basic standard requests ........................................ 48
7.4 Specific device driver ........................................... 49
  7.4.1 General functions ........................................... 50
  7.4.2 Specific functions ......................................... 51
7.5 HUB driver ..................................................... 51
7.6 USB controller .................................................. 53
  7.6.1 USB controller driver ....................................... 55
7.7 Detecting a device plug in ...................................... 63

8 Implementation’s design ......................................... 65
  8.1 Design overview ............................................... 65
  8.2 Initialize EHCI controller driver ............................. 67
  8.3 Gain ownership of the EHCI controller ....................... 68
  8.4 Initialize EHCI controller ..................................... 68
    8.4.1 Reset the EHCI controller ................................. 69
    8.4.2 Basic configuration of the controller .................... 70
  8.5 Initialize the general USB driver ............................ 70
    8.5.1 Set up the PORTSC structure ............................ 71
    8.5.2 Set up the transfer class ................................ 72
    8.5.3 Set up the Hub class ..................................... 72
  8.6 Scan port until we detect a new device plugged .............. 74
  8.7 Initialize device ............................................ 75
  8.8 Load suitable driver(Keyboard driver) ....................... 76
  8.9 Configure device ............................................. 76

9 Economic valuation ............................................... 77
  9.1 Time distribution among tasks comparison .................... 77
  9.2 Real budget distribution ..................................... 80
  9.3 Future work .................................................. 80
List of Figures

Figure 4.1  Time distribution among tasks  . . . . . . . . . . . . . . 17
Figure 4.2  Gantt chart (1)  . . . . . . . . . . . . . . . . . . . . . 20
Figure 4.3  Gantt chart (2)  . . . . . . . . . . . . . . . . . . . . . 21
Figure 4.4  Gantt chart (3)  . . . . . . . . . . . . . . . . . . . . . 21
Figure 5.1  Budget distribution among tasks  . . . . . . . . . . . . 26
Figure 7.1  Keyboard connected to a computer scheme  . . . . . . 32
Figure 7.2  USB topology with a root hub (as the physical port) and  
some devices and hubs connected.  . . . . . . . . . . . . . . . . . . 35
Figure 7.3  Packet structure  . . . . . . . . . . . . . . . . . . . . . 37
Figure 7.4  Out transaction set of packets. Purple is a packet from  
host to device. Green is a packet from device to host.  . . . . . . . 40
Figure 7.5  In transaction set of packets. Purple is a packet from  
host to device. Green is a packet from device to host.  . . . . . . . 42
Figure 7.6  Setup transaction set of packets. Purple is a packet  
from host to device. Green is a packet from device to host.  . . . . 42
Figure 7.7  Stages of a device  . . . . . . . . . . . . . . . . . . . . . 45
Figure 7.8  Structure of the project (1). Red used to represent soft-
ware. Purple used to represent hardware.  . . . . . . . . . . . . . . 49
Figure 7.9  Structure of the project (2). Red used to represent soft-
ware. Purple used to represent hardware.  . . . . . . . . . . . . . . 51
Figure 7.10  *Structure of the project (3). Red used to represent soft-
ware. Purple used to represent hardware.* .......................... 53

Figure 7.11  *Structure of the project (4). Red used to represent soft-
ware. Purple used to represent hardware.* .......................... 55

Figure 7.12  *Structure of the project (5). Red used to represent soft-
ware. Purple used to represent hardware.* .......................... 56

Figure 7.13  *Set of registers base adress* .............................. 57

Figure 7.14  *OS controller ownership state diagram* .................... 60

Figure 8.1  *Hub basic structures* .............................. 73
Chapter 1

Glossary

This chapter of the project will briefly describe key concepts necessary for this project comprehension.

- **Universal Serial Bus (USB)**: it defines the cables, connectors and communication protocols used in a bus for communication, power supply and connection between computer and electronic devices.

- **Packet Identifier (PID)**: in a transport layer each table or elementary stream is identified by a number. This number is called packed identifier.

- **End of Packet (EOP)**: specific stream of bits identifying the end of a Packet.

- **Acknowledgement (ACK)**: signal between processes to acknowledge that previous message has been received.

- **Host**: main device where all the other devices are connected. In our case the laptop.
• **Client**: device connected to the Host. In our case it will be the keyboard.
Chapter 2

Context and state-of-the-art

This chapter purpose is to introduce the topic and the context of the project, define some of the terms and concepts under study, explain the technology used and review similar projects.

2.1 Context

2.1.1 Relevance and motivation

Communication protocols have always been present during the technology revolution but, over time, some protocols have gone a step forward and become a standard. The USB (short for Universal Serial Bus) has been a communication protocol standard present during the past 20 years [1]. During these years it has evolved to adapt to the new technologies and their growth of speed.

The USB protocol was initially designed to standardize the connection of
computer peripherals as keyboards, printers or mouses but it has become commonplace on other devices such as video game consoles and smart phones. This wide implementation is due to the versatility that the protocol offers and the adaptation to the new necessities the technology has required during the years.

In order to have this kind of versatility the protocol has reached high levels of complexity. To facilitate programmers the job of creating USB drivers several libraries have been deployed with all the necessary tools to make it easy for the developer to code the driver.

The complexity of the protocol then resides on these libraries which implement all the functionality the USB protocol has. Although it can be an advantage for most of the coders to abstract the intricacies of the protocol, it is a disadvantage if a specific small driver has to be developed, since the full library will be needed even if only a small part of it is used.

The main motivation of this project is, in the first place, to describe in a comprehensive way how the USB protocol work to help future computer engineers or computer enthusiasts have an overall view of the protocol. In the second place, help them write it’s own library or minimal driver (without any external library) to work with a specific component.

2.1.2 Objectives and problem formulation

Once the context has been understood we can define the problem in the following way. We need to describe the protocol from the ground up in order to facilitate the comprehension and the abstraction of functionality. This will allow us to isolate different parts of the protocol and minimize the
This solution will include a rigorous dissection of the protocol, explaining it in detail from the ground up to the leaves. This dissection will be used later to implement, a minimal example of a USB driver (without the use of any external libraries but the driver itself). The purpose of implementing this example step by step is to show how a driver for a specific functionality could be built. In our case the driver will be able to communicate a specific keyboard with a precise operating system.

Finally, once the driver is implemented and thoroughly documented, future computer engineers and computer enthusiasts could use it as a base for their own implementations. Although the final purpose of their driver may be different from the example presented here, the protocol dissection, the example driver implementation and its step by step explanation may be followed in the creation of new specific implementations.

The objectives of the project are:

1. Perform a reverse engineering of the USB protocol in order to describe and document in a comprehensive way how the protocol works and a precise dissection of it.

2. Step by step implementation example of a driver capable of enabling the communication between a specific keyboard with a certain operating system.
2.1.3 Actors

In this section we will briefly identify the interested parties interested in this project, who will use it and benefit from it.

- **Developer of the project:** he has the interest in achieving a good result for the project. His goal is to provide the other actors with a usable, functional documentation and examples.

- **Project director and co-director:** this project will be useful for both the director and the co-director to improve other projects they are involved with.

- **Computer engineer students:** the fact that this project will be presented as a document stating and analysing each part of the USB protocol may help future students on the development of similar projects.

- **Companies:** this project will be uploaded to the web and will be accessible for anyone to review the work done. This include companies interested in developing projects over USB protocol. They can benefit on the work done in this project for they own profit.

- **Computer enthusiasts:** as explained for companies anyone interested in increasing the boundaries of his/her knowledge is welcome to read this project and benefit from it.


## 2.2 State-of-the-art

### 2.2.1 Technology used

The technology used is a specific keyboard and a Unix based operating system. The keyboard must remain the same for the whole transition of the project, since we want to implement the minimum code for the driver to work we need to hard-code some parameters. Using the same keyboard from start to end will allow us to perform a reverse engineering of the significant bit streams it uses and save them for the explanation/example.

The operating system must be, as the keyboard, the same for all the project. Note that this doesn’t mean that if a developer tries to implement a USB driver or USB-compatible library by following this documentation, they would have to use the same operating system described herein. For the example we are going to use the CentOS [4] Operating system.

### 2.2.2 Similar projects

A good documentation of the USB protocol could be the beyondlogic wiki [5] that contains a USB protocol explanation project with lots of information. This project has helped me understand the main part of the protocol.

During the research part we found a similar project from Alex Chadwick [6] where he implements a USB driver for a raspberry pi. This project will allow us to understand the basics of the driver implementation.

Although the goal of this document is totally different, a similar project could
be the development of a shadow keylogger. The purpose of this is to record
the keystrokes a keyboard does in a computer without the owner knowing it.
This product has a place in the market and the price oscillates between the
20€ and 100€. As you might imagine the ethic behind a project like this is
confuse.
Chapter 3

Scope of the project and methodology

The following chapter will explain the boundaries of the project, the methodology I am going to use, how we are going to organize and prioritize the tasks and how to validate these tasks.

3.1 Project boundaries

As the technology used will not be a problem to acquire, the only boundary I may find is the available time. Therefore, after the protocol and the minimal driver have been done, different extensions to generalize the implementation will be added (as many as the time permits). This could allow a better understanding for the reader as having two examples (or an extended example) could give him/her a better understanding of how to implement its
3.2 Methodology

The methodology chosen for this project has been SCRUM - Agile [7] with different iterations with well defined tasks. This system will allow us to have a first approach to the minimal driver and re-implement it in future iterations. The use of Agile methodology will be effective to modify certain parts of the implementation while we are implementing a second example if bugs are found without affecting the work plan.

3.3 Tools

Although the project is being developed by one person it is important to establish deadlines for the different tasks. Organization is a priority to end the project in time.

As a tracking tool I am going to use the web application Trello [8]. This tool will be used to state deadlines, elaborate tasks to be done, obtain feedback from the director and co-director and revise finished tasks.

As code is going to be a part of this project I will use Github [9] as a repository. The repository offers me a mobility that other off-line repositories don’t offer. Also, if a roll-back in the code is needed we will be able to do so in an easy way. If we have time to develop a second example based on the first one we will be able to fork the first one and don’t lose anything. As mentioned before, this project will be available for anyone who wants to use
it (in other words, open source). For this reason, Github will be a useful tool to have the code available for anyone who needs it.

### 3.4 Validation methodology

Through the course of the project, several meetings with the director and co-director of the project will be held. These meetings will have the purpose of revising and validating the implementations and documentation present in this project.

Also, during the GEP course some of the writing part will be submitted to my professor who will give me feedback on how to improve this same part.
Chapter 4

Project planning

In this chapter the different tasks performed during the project will be explained. These tasks will be split in separated blocks and chronologically ordered using the Gantt diagram attached at the end of the chapter. Alternatives to the original plan will be detailed and how the project will be modified if any task fails. Also, the different resources used during the project will be explained.

4.1 Project duration

The project will have a total duration of five months starting in September 2016 and ending in January 2017. This period fits in the fall semester of UPC and the core of the project will be developed during the months of October, November and December. September will be used to study the idea of the project and structure it. January will be used to revise the document and to
prepare the presentation.

4.2 Resources

The resources used in the project can be divided into two categories:

- **Human resources**: the project will be done by one person developing the tasks previously explained. Two other persons will be supervising the work.

- **Material resources**: the resources needed for this project will be: a personal computer, a physical USB keyboard, a Linux distribution operating system, Github and Trello.

4.3 Gantt diagram and tasks description

The Gantt diagram is based on the methodology Agile, meaning that the distinct tasks represented on it will follow different sprints. Furthermore, the diagram is divided in two big blocks:

1. The different assignments presented in the GEP course and the work related to the implementation.

2. The documentation of the project itself.

The tasks are described below:
• **GEP:** this part will present the delivery deadlines for the different assignments of the class *Gestió de Projectes*. The utility of these deliveries is to define, as much as possible, the project on its first stage. This is the reason why this part is situated just before starting to implement the project. The study of this part is strictly related with the preparation of the environment and the research of the pre-requirements. The whole set of tasks will be done by the developer.

Different tasks and the deadlines:

- Deliverable 1: context and scope of the project 27/09/2016 (25 hours). No requirements are needed. The laptop will be needed to develop this task.

- Deliverable 2: project planning 03/10/2016 (9 hours). Deliverable 1 must be finished before starting the deliverable 2. The laptop and the web application Trello will be used to develop this task.

- Deliverable 3: budget and sustainability 10/10/2016 (10 hours). Deliverable 2 must be finished before starting the deliverable 3. The laptop will be needed to develop this task.

- Deliverable 4: first oral presentation 17/10/2016 (7 hours). Deliverable 3 must be finished before starting the deliverable 4. The laptop will be needed to develop this task.

- Deliverable 5: oral presentation, *bloc d’especialitats* and final document 24/10/2016 (40 hours). Deliverable 4 must be finished before starting the deliverable 5. The laptop will be needed to develop this task.

• **Approach to USB protocol:** this part consists on the research and study of documentation stating how the USB protocol works. The
purpose of this part is to reach a full understanding of the core of the USB protocol. This knowledge will be used in future parts of the project to design and implement the driver.

This is formed by these two sub-tasks:

- Research of USB protocol information (10 hours). The first two GEP deliverables must be finished before starting this task. The laptop will be needed to develop this part.
- Study of the protocol (30 hours). The research of the USB protocol information must be done before starting this task. The laptop will be needed to develop this part and it will be done by the developer.

• **Design of USB Driver:** once the protocol is well comprehended we will proceed to design, in a conceptual way, how the driver will be organized. The design needs to be as precise as possible to make the next task (implementation) as smooth as possible. These set of tasks will be done by the designer. The Approach to USB protocol should be done before starting these sub-tasks. The laptop will be needed for all the sub-tasks.

The sub-tasks are:

- Pairing the device (40 hours).
- Pulling and retrieving data (20 hours). The sub-task *pairing the device* should be finished before starting this other sub-task.

• **Implementation:** as we explained in the previous task (design) the implementation will follow what has been designed. Once we have a
first approach to the implementation it will be tested. If this implement-
mentation does not work as expected we will go back to designing and
re-implementing. These set of tasks will be done using the laptop, the
keyboard, and the Github repository. All the sub-tasks will be devel-
oped by the programmer.

The following list exposes the sub-tasks:

- Implementing the part *Pairing to a device* (70 hours).
- Implementing the part *Pulling and retrieving data* (50 hours). The
sub-task *pairing to a device* should be finished before starting this
sub-task.
- Testing (30 hours). The sub-task *pulling and retrieving data* should
be finished before starting this sub-task.
- Documentation (30 hours). The sub-task *testing* should be fin-
ished before starting this sub-task.

Time distribution among tasks can be visualised in Figure 4.1 and a complete
summary including sub-tasks can be found Table 4.1.
Figure 4.1: Time distribution among tasks
<table>
<thead>
<tr>
<th>Task</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total GEP</td>
<td>70</td>
</tr>
<tr>
<td>Deliverable 1</td>
<td>25</td>
</tr>
<tr>
<td>Deliverable 2</td>
<td>9</td>
</tr>
<tr>
<td>Deliverable 3</td>
<td>10</td>
</tr>
<tr>
<td>Deliverable 4</td>
<td>7</td>
</tr>
<tr>
<td>Deliverable 5</td>
<td>19</td>
</tr>
<tr>
<td>Approach to USB protocol</td>
<td>40</td>
</tr>
<tr>
<td>Research of USB protocol and information</td>
<td>10</td>
</tr>
<tr>
<td>Study of the protocol</td>
<td>30</td>
</tr>
<tr>
<td>Design of the USB driver</td>
<td>60</td>
</tr>
<tr>
<td>Pairing the device</td>
<td>40</td>
</tr>
<tr>
<td>Pulling and retrieving data</td>
<td>20</td>
</tr>
<tr>
<td>Implementation</td>
<td>180</td>
</tr>
<tr>
<td>Implementing the part <em>Pairing to a device</em></td>
<td>70</td>
</tr>
<tr>
<td>Implementing the part <em>Pulling and retrieving data</em></td>
<td>50</td>
</tr>
<tr>
<td>Testing</td>
<td>30</td>
</tr>
<tr>
<td>Documentation</td>
<td>30</td>
</tr>
<tr>
<td>Total</td>
<td>350</td>
</tr>
</tbody>
</table>

Table 4.1: Time estimation
4.4 Alternatives and action plan

The fact that we are working with the methodology Agile allows us to iterate through the different parts until an expected result is obtained. If less time than scheduled is needed to arrive to the final result this extra time is assigned to the next task.

To deal with punctual problems more time than expected has been assigned to each task. This will allow us to consume time if several iterations need to be done.

The tasks (as implementation or design) that could present more difficulties have been assigned with time for an extra iteration if needed. This tasks are mentioned below:

- **Design of USB Driver**: the time assigned to the task is 150% the expected time. This task may be extended to a 200% if we don’t obtain results on the second sprint. This will be 20 extra hours.

- **Implementation**: the time assigned to the task is 150% the expected time. This task may be extended to a 200% if we don’t obtain results on the second sprint. This will be 40 extra hours.

- **Testing**: the time assigned to the task is 150% the expected time. This task may be extended to a 200% if we don’t obtain results on the second sprint. This will be 10 extra hours.
4.5 Gantt diagram

This section purpose is to illustrate the project schedule. The start and finish dates of the different sub-tasks are specified on the chart and colors are used to identify different tasks easily.

Figure 4.2: Gantt chart (1).
Figure 4.3: Gantt chart (2).

<table>
<thead>
<tr>
<th>Etapas proyecto</th>
<th>Noviembre 2016</th>
<th>Diciembre 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Research in USDU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Study of research</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design of USD...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Placing the device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulling and refill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Placing the device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulling and refill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4: Gantt chart (3).

<table>
<thead>
<tr>
<th>Etapas proyecto</th>
<th>Enero 2017</th>
<th>Febrero 2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliverable 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Research in USDU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Study of research</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design of USD...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Placing the device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulling and refill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Placing the device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulling and refill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Budget

The following chapter purpose is to identify the different types of costs associated to the project, organise them and present a final budget. For a better comprehension costs have also been distributed among the different tasks presented in the previous chapter. Lastly, an explanation of how the project will be controlled and a study explaining how we are going to treat unforeseen contingencies is presented.

5.1 Identification of costs

In this section we will see the different costs divided into three categories: human, hardware and software.
5.1.1 Human costs

For the appropriate development of the project three people will be needed, a developer and two directors of the project.

The following criteria has been established to calculate the total number of working hours for this project for each role. The developer of the project will have the total hours established in the Gantt diagram as he will be the responsible for those tasks in the project. To calculate the total hours of the directors an estimation of the time spend in weekly meetings has been used.

The cost per hour established in the Table 5.1 is not real but an estimation. The final price for the humans costs is 14.150 €.

<table>
<thead>
<tr>
<th>Role</th>
<th>Salary per hour (€/h)</th>
<th>Dedication Time</th>
<th>Final Cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>25</td>
<td>350 hours</td>
<td>8750</td>
</tr>
<tr>
<td>Director 1</td>
<td>45</td>
<td>60 hours</td>
<td>2700</td>
</tr>
<tr>
<td>Director 2</td>
<td>45</td>
<td>60 hours</td>
<td>2700</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>-</td>
<td>14.150</td>
</tr>
</tbody>
</table>

Table 5.1: Human costs

5.1.2 Hardware costs

The hardware used in this project is an Asus F541UA-XX054T (laptop) and a Logitech Keyboard K120. Both devices are recycled from other projects. The amortization is calculated based on the lifetime of the hardware and the portion of time used to develop this project. The resulting costs can be consulted in the Table 5.2.
<table>
<thead>
<tr>
<th>Equipment</th>
<th>Cost (€)</th>
<th>Amortization</th>
<th>Project cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asus laptop</td>
<td>479</td>
<td>75%</td>
<td>119</td>
</tr>
<tr>
<td>Logitech keyboard</td>
<td>13</td>
<td>90%</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>120</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Hardware costs

5.1.3 Software costs

As mentioned before the different software used in this project is open source with free licence so the total cost is 0 €. A list of the software used is given in Table 5.3.

<table>
<thead>
<tr>
<th>Software</th>
<th>Licence (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentOS</td>
<td>0</td>
</tr>
<tr>
<td>Trello</td>
<td>0</td>
</tr>
<tr>
<td>Sublime Text</td>
<td>0</td>
</tr>
<tr>
<td>Gmail</td>
<td>0</td>
</tr>
<tr>
<td>Github</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

Table 5.3: Software costs

5.1.4 Final budget

The final budget is composed by the software costs, the hardware costs and the human costs. The Table 5.4 shows a summary and the final budget.
The taxes for this project will be a 21%. The percentage must be added to the final budget thus, the total cost of the project is: **17.267€**

## 5.2 Budget distribution

The budget is distributed among the four main task groups in the following way:

- **GEP**: uses 20% of the human costs and the 25% of the laptop time. The total cost of GEP is: **2.859€**.

- **Research and study**: uses 11% of the human costs and the 25% of the laptop time. The total cost of research and study is: **1.586€**.

- **Design**: uses 18% of the human costs and the 25% of the laptop time. The total cost of design is: **2.576€**.

- **Implementation**: uses 51% of the human costs, 25% of the laptop time and 100% of the keyboard time. The total cost of implementation is: **7.247€**.
Take in mind a 21% of tax needs to be added in the budget distribution.

Figure 5.1 represents the budget distribution among the different tasks.

Figure 5.1: Budget distribution among tasks

5.3 Control management and unforeseen contingencies

5.3.1 Control management

A margin of error was included in the Gantt diagram to solve minor problems. Certain number of minor problems are included in the budget already presented. If the number is over passed a new budget will be calculated. See unforeseen contingencies for more information.
If bigger problems were found compromising the deadline or the budget, as changing the prerequisites of the whole project, a new budget will be calculated.

5.3.2 Unforeseen contingencies

To adjust in each iteration the time, we are going to establish a control of the hours the developer of the project does. If an iteration is finished with less time than estimated, this time will be added to the next task or iteration. The opposite will be done in the other case.

Three main tasks may last longer than expected. This is explained in the section 4.5. The resulting study says that 70 developing hours may be added if the project does not go as expected. It may increment the final budget in 1.750€. 
Chapter 6

Economic, social and environmental sustainability

This chapter purpose is to detail whether or not the project is sustainable. To do so we will present three studies: economic, social and environmental. Finally, with the sustainability matrix attached at the end of this chapter we will summary the results obtained.

6.1 Economic dimension

Costs of the project have been estimated among human and material resources in the chapter Identification of costs. In order to reduce the economic impact free licence software and recycled hardware has been used for the development. Because of the reasons above the economic project development mark is 9 (0:10).
The more similar project, as explained in the chapter *Context and state-of-the-art*, are shadow keylogger with a value in the market of 20€. This may allow users develop this kind of projects without any cost. Although this can save lots of money to people willing to use this type of software, this project aims to be more ethic and is not willing people to use this document to implement illegal software. Because of the reasons above the economic exploitation mark is 7 (0:20).

This project is prepared with a budget control management and unforeseen contingencies have been explained. Because of this the mark given to economic risks is -5 (-20:0)

### 6.2 Social dimension

The realization of this project will bring the developer the satisfaction of helping the Internet community documenting how the USB protocol work and giving some examples. For this reason the mark given to social project development is a 8 (0:10).

Although we can find similar projects as shadow keyloggers these usually have a paid-licence and are not open source. Also, the documentation of the USB protocol is not abundant. The aim of this project is then, to document and give an implemented example in the same project for others to learn. Because of the reasons above i state there is a real need for the project and therefore the social exploitation mark is a 12 (0:20).

This project audience is expected to be computer engineer students or enthusiasts. Although without this project the audience may not be in a weak
position of knowledge they may find, in this project, a friendly documentation and a step by step example allowing them to learn quicker. This is why the social risks mark is -10 (-20:0).

6.3 Environmental dimension

The environmental footprint of the project is quite small because the only hardware devices used are a keyboard and a laptop (both recycled). Although I’ve tried to reduce the footprint to the minimum this can be reduced even more by using older computers or eco-friendly hardware. That is the reason why the environmental project development mark is 7 (0:10).

Once the development of this project is finished custom USB drivers will be implemented with less code lines. This will allow CPUs to run the same functionality with less code lines and therefore power consumption may be reduced. This is why the environmental exploitation mark is 12 (0:20).

The only scenario where the footprint may be incremented is if the keyboard or the laptop broke down and need a replacement. The environmental risks mark is then -2 (-20:0).

6.4 Sustainability summary

To summarize the previous sections Table 6.1 presents environmental, economic and social marks in a matrix. The sum of the different marks give us a total of 38 (-60:90). Therefore, we can state our project is sustainable.
## Table 6.1: FDP Sustainability Matrix

<table>
<thead>
<tr>
<th></th>
<th>Project Development (0 to 10)</th>
<th>Exploitation (0 to 20)</th>
<th>Risks (-20 to 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environmental</td>
<td>7</td>
<td>12</td>
<td>-2</td>
</tr>
<tr>
<td>Economic</td>
<td>9</td>
<td>7</td>
<td>-5</td>
</tr>
<tr>
<td>Social</td>
<td>8</td>
<td>12</td>
<td>-10</td>
</tr>
<tr>
<td>Sustainability range</td>
<td>24</td>
<td>31</td>
<td>-17</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>38</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: FDP Sustainability Matrix
Chapter 7

USB reverse engineering

The purpose of this chapter is to make the reader understand the USB protocol. We will start introducing the terminology and basic concepts necessary to understand the whole process. Then, a summary of the whole project will be present to give the reader an abstract perspective of what we want to achieve. Later an explanation of the communication process will be introduced to finally detail the different layers involved in the USB protocol.

7.1 Introduction

As described before the universal serial bus (USB) was the solution developed by several institutions to satisfy the basic requirement of plug-and-play. The
aim was to find a solution to the mixture of connection methods to the PC (serial ports, parallel ports, keyboard and mouse connections and so on) and that no user-knowledge would be required. The purpose was to load a suitable driver automatically when a device is discovered.

On the following sections we are going to walk through the different parts of the USB protocol to be able to communicate a keyboard device with a computer (Figure 7.1).

For the aim of this project we decided to follow the current specification ”Universal Serial Bus Specification, Revision 2”. The specification can be downloaded from the official USB web page [11]. Before trying to read it take in mind its more than a thousand pages of dense information (60MB).

Although we can find all the information regarding USB 2.0 in the specification we have extracted some relevant parts of the USB made simple web page [12] to document this project. If more information of one of the following sections is needed more detailed information can be found in this website.

Only when the specification is understood we will move to the next objective, implement the minimal keyboard driver.

The specification covers the three different data speeds supported by USB:

- Low speed
- Full speed
- High speed
7.1.1 Key concepts

Let us introduce to you to some key concepts about the USB protocol before we start the next sections:

- **Topology**: USB protocol uses a star topology. There is just one host (the PC in our case) and several devices connected to this host.

- **Address**: all the devices will have a unique address. This will be assigned by the host at the appropriate time (explained accurately on the next sections). The number of devices connected to a host is limited due to the number of bits reserved on the packets transmitting the information. Seven bits are reserved for this purpose and address zero is reserved for special cases. The total number of devices connected to a host is 127.

- **Endpoints**: each device will have two or more endpoints. These will serve as a sink of data (IN or OUT). The number of endpoints each device has will be described by the same device when connected (explained in the following sections).

- **Communications**: all communications will be started by the host, preventing the devices to communicate directly and the devices will have one or more endpoints and each is a sink data (IN or OUT). Endpoint 0 is also a special case and it is used to control the device.

- **Software hub**: although the total number of devices connected to a host can be 127 normally the host will not have more than 3-4 USB ports. Using a software hub we will transform a port to a hub where several devices and/or hubs can be connected to. Figure 7.2 shows an
example of the USB topology with the root hub (the host) and some devices and hubs connected.

Figure 7.2: USB topology with a root hub (as the physical port) and some devices and hubs connected.

7.1.2 General idea

The USB protocol was designed to communicate the host with any device connected to its ports. For this reason, the communication protocol must always remain the same. Section 7.2 Data flow will approach the protocol explaining the different types of packets, transactions and transfers available and its purposes.

Once we understand the communication protocol we will continue explaining the core of the USB protocol, how to own a device and set it up. This steps will be the same for all the devices and the core driver will be in charge
of them. Each device will have an associated driver capable to respond to the device specific functionality. This driver will be built on top of the core driver. To allow different devices connect to the same port (through a hub) we will need a software hub implemented also on top of the core driver. Finally, the core driver will need to communicate with the port itself and to do so we will build a driver capable of reaching the controller’s registers.

7.2 Data flow

In this section the different type of packets used for the communication will be explained but first, two important concepts where specified in the previous section:

- USB topology is based on a star topology
- All communications must be always initialized by the host.

At any point of time only the host or one device may be transmitting packets. All packets sent by the host will be distributed to all the devices and only when a packet reaches its addressee this will be permitted to send back the reply.

7.2.1 Packets

The communication between the host and the device is done through packets. The fields of these packets are in a little-endian format and it always start with a sync pattern to allow the receiver bit clock to synchronize, followed by
the data bytes and a end of packet (EOP). Figure 7.3 illustrates the common packet structure.

![Packet structure](image)

Figure 7.3: *Packet structure*

Packets are divided into four sets depending on their purpose:

- Token
- Data
- Handshake
- Special

On the following paragraphs we are going to describe the main structure of tokens, data, handshake and special packets. The first byte in every packet will be used to determine what kind of packet is. This identifier is called packet identifier (PID) and Table 7.1 illustrates the different values it can take. A brief summary of the functionality the sets have:

- **Token packet**: always the first packet in a transaction. It will contain the reason of the transaction and the endpoint it is addressed. Table 7.2 illustrate the different fields present in a token packet. A cyclic redundancy code (CRC) of 5 bits (CRC5) will be used to validate the reception of the data.

- **Data packet**: used if a transaction has a data stage. The field structure used is shown in Table 7.4. A cyclic redundancy code (CRC) of 16 bits (CRC16) will be used to validate the reception of the data.
<table>
<thead>
<tr>
<th>PID type</th>
<th>PID name</th>
<th>PID bits 3:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>OUT</td>
<td>0001b</td>
</tr>
<tr>
<td>Token</td>
<td>IN</td>
<td>1001b</td>
</tr>
<tr>
<td>Token</td>
<td>SOF</td>
<td>0101b</td>
</tr>
<tr>
<td>Token</td>
<td>SETUP</td>
<td>1101b</td>
</tr>
<tr>
<td>Data</td>
<td>DATA0</td>
<td>0011b</td>
</tr>
<tr>
<td>Data</td>
<td>DATA1</td>
<td>1011b</td>
</tr>
<tr>
<td>Data</td>
<td>DATA2</td>
<td>0111b</td>
</tr>
<tr>
<td>Data</td>
<td>MDATA</td>
<td>1111b</td>
</tr>
<tr>
<td>Handshake</td>
<td>ACK</td>
<td>0010b</td>
</tr>
<tr>
<td>Handshake</td>
<td>NAK</td>
<td>1010b</td>
</tr>
<tr>
<td>Handshake</td>
<td>STALL</td>
<td>1110b</td>
</tr>
<tr>
<td>Handshake</td>
<td>NYET</td>
<td>0110b</td>
</tr>
<tr>
<td>Special</td>
<td>PRE</td>
<td>1100b</td>
</tr>
<tr>
<td>Special</td>
<td>ERR</td>
<td>1100b</td>
</tr>
<tr>
<td>Special</td>
<td>SPLIT</td>
<td>1000b</td>
</tr>
<tr>
<td>Special</td>
<td>PING</td>
<td>0100b</td>
</tr>
<tr>
<td>Special</td>
<td>Reserved</td>
<td>0000b</td>
</tr>
</tbody>
</table>

Table 7.1: PID type and names

- **Handshake packet**: used when a status stage need to be transmitted. Table 7.3 illustrates the field structure.

- **Special packet**: these packets are used in special cases as for example managing errors.
7.2.2 Transactions

As we have said before packets are used in transactions. These are the three types of transactions:

- Out transaction
- In transaction
- Setup transaction

For a better comprehension of these transactions we have this definitions:

- **Out transaction**: consists of two or three packets depending on whether the transaction needs a handshake packet or not (depending if it is an Isochronous transfer or not, this will be explained on the next paragraphs). This transaction (illustrated in Figure 7.4) consists of:
Figure 7.4: *Out transaction set of packets.* *Purple is a packet from host to device. Green is a packet from device to host.*

1. Host to device token packet.
2. Host to device data packet.
3. Device to host handshake packet (not always).

- **In transaction:** similar to out transaction depending on whether the transaction needs a handshake packet or not it will consist of two or three packets. This transaction (illustrated in Figure 7.5) consists of:

  1. Host to device token packet.
  2. Device to host data packet.
  3. Host to device handshake packet (not always).

- **Setup transaction:** A successful setup transaction consists of three sequential packets. This transaction (illustrated in Figure 7.6 and Table 7.5) consists of:

  1. Host to device token packet.
  2. Host to device data packet (exactly 8 bytes).
  3. Device to host handshake packet.
<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reqType</td>
<td>1</td>
<td>bitmap</td>
<td>D7 data direction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Host-to-device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Device-to-host</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>D6:5 Type</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Standard</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 = Vendor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 = Reserved</td>
</tr>
<tr>
<td>1</td>
<td>bRequest</td>
<td>1</td>
<td>value</td>
<td>specific req</td>
</tr>
<tr>
<td>2</td>
<td>wValue</td>
<td>2</td>
<td>value</td>
<td>variables according req</td>
</tr>
<tr>
<td>4</td>
<td>wIndex</td>
<td>2</td>
<td>index or offset</td>
<td>variables according req</td>
</tr>
<tr>
<td>6</td>
<td>wLength</td>
<td>2</td>
<td>count</td>
<td>n° of bytes in data stage</td>
</tr>
</tbody>
</table>

Table 7.5: Setup transaction fields specification, the different standard bRequests can be found in Table 7.6.
Figure 7.5: In transaction set of packets. Purple is a packet from host to device. Green is a packet from device to host.

Figure 7.6: Setup transaction set of packets. Purple is a packet from host to device. Green is a packet from device to host.
7.2.3 Transference of data

There are four different ways to transfer data on a USB bus and one or more transaction types may be used by each one. These are:

- Control transfer
- Bulk transfer
- Interrupt transfer
- Isochronous transfer

Here I will present a brief explanation of them:

- **Control transfer**: used for initial configuration of the device by the host, using endpoint 0 IN and OUT. As we have explained before these endpoints are specifically reserved for this purpose. If required by the device, these endpoints may be used after configuration as part of the device-specific control protocol. Control transfer consists of three different stages:

  1. **Setup stage**: is used to define the request and specify whether and how much data should be transferred in the next stage. It carries 8 bits called the setup packet.

  2. **Data stage**: may or may not be present depending on the information provided in the setup stage. It always starts sending a DATA1 packet and then it will alternate between DATA0 and DATA1 packets until all the required data has been transferred.
3. **Status stage:** contains just one packet of type DATA1 empty. Status stage transfer direction is always opposite to data stage (host to device or device to host).

- **Bulk transfers:** main purpose is to transfer large amounts of data with error-free delivery. The bandwidth is not guaranteed in these type of transfers and for this reason the host will schedule bulk transfers after allocating other transfer types. For these transfers DATA0 packets are used.

- **Interrupt transfers:** although the name could suggest these type of transfers uses interrupts this is false. The name is chosen because interrupts would have been used in early connection types for the same purpose. They are used when the host needs to keep up to date with of any changes of status in a device (e.g. mouse or keyboard). It consists of three steps:
  1. Host to device token packet
  2. Device to host data packet
  3. Host to device handshake packet

- **Isochronous transfer:** is the opposite to bulk transfers. Guarantees bandwidth but does not guarantee error-free delivery. The main purpose is to serve data that needs to maintain a data flow and where it’s not important if some data gets lost. As we don’t care about the handshake process at the end of transaction these transactions will just have two packets.
7.3 Controlling a device

The purpose of this section is to explain the different stages of a device. How we can move from one to the other and the peculiarities of each one of them. All the devices have three different stages (illustrated in figure 7.7):

1. **Powered**: in this stage the device has just been powered up or performed a reset. It will respond to address 0 and only control transfers are allowed (endpoint 0).

2. **Addressed**: in this stage the device already has an address. The host must request information to the device during this stage to set up the appropriate configuration. During this stage only control transfers are allowed (endpoint 0).

3. **Configured**: once a configuration has been chosen the device will be in a configured stage. In this stage the device can start working as the device was designed to be.

![Stages of a device](image)

Figure 7.7: *Stages of a device*

During the first two stages all devices must respond to the standard requests specified in the specification. Table 7.6 illustrates all the standard requests the host can send. On the configured stage the device will have to respond to standard requests and the specifics each device has.

When a device is plugged in, the host needs to perform some actions in order...
to go through the different stages until it reaches the configured stage. This are the steps to be followed:

1. **Perform a reset to the device.** Returning a device to a known stage (powered) is necessary before starting to configure it. One device at a time will be reset by the host so that there is no danger of two devices responding at the same address 0.

2. **Get the maximum packet size supported.** We can get this information sending a *get descriptor* command to the device. The response to this get may be quite long. For this reason we may want to reset the device (a second time) and continue with addressing.

3. **Perform a second reset to the device.** This step is not necessary. It will depend on whether we want to wait for the device to finish sending all the data we have requested or not.

4. **Set a unique address.** The host is the only one addressing the devices so it must save all the used addresses. A not-used address will be set to the new device using the *set Address* command. Once the address is set the device change to addressed state (Figure 7.7)

5. **Get information from the device.** At this point the host must determine what kind of device we have. This information will be used to load a suitable device driver. Information requested to the device may be:

   - What kind of device are you?
   - What company produced you?
   - Do you have several configurations? if so, enumerate them.
All this requests will be asked through the standard requests illustrated in Table 7.6.

6. **Load a suitable driver.** Once the general driver find a suitable driver, this will be loaded. Once again the driver will communicate with the device through the standard requests (Table 7.6) to verify if it meant to control the device.

7. **Select the appropriate configuration.** The device driver will select the most appropriate configuration for the device. Once this is done the device will change to configured stage (Figure 7.7). We will select the appropriate stage using the *set Configuration* command.

<table>
<thead>
<tr>
<th>Request</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Feature</td>
<td>01h</td>
</tr>
<tr>
<td>Get configuration</td>
<td>08h</td>
</tr>
<tr>
<td>Get descriptor</td>
<td>06h</td>
</tr>
<tr>
<td>Get interface</td>
<td>0Ah</td>
</tr>
<tr>
<td>Get status</td>
<td>00h</td>
</tr>
<tr>
<td>Set address</td>
<td>05h</td>
</tr>
<tr>
<td>Set configuration</td>
<td>09h</td>
</tr>
<tr>
<td>Set descriptor</td>
<td>07h</td>
</tr>
<tr>
<td>Set feature</td>
<td>03h</td>
</tr>
<tr>
<td>Set interface</td>
<td>0Bh</td>
</tr>
<tr>
<td>Sync frame</td>
<td>0Ch</td>
</tr>
</tbody>
</table>

Table 7.6: Set of standard requests
7.3.1 Basic standard requests

For a better comprehension we will briefly explain the standard requests used in the project:

- **Get descriptor:** this request will allow us to detect the maximum packet size during the powered stage. It will later be used to know what kind of device it is (e.g. keyboard). Among the information that can be requested with this descriptor we have:
  - Maximum packet size for endpoint 0.
  - USB specification release number (e.g. USB 2.0).
  - Vendor identification.
  - Product identification.
  - Number of configurations.

- **Set address:** the host must have a record of the different devices connected at all time. It also have a record of the address of each device so all the devices have a unique address. When the host feels it’s time to go to the addressed stage it sends the device the set address command. The device then has 2 ms to get ready to respond to the new address.

- **Set configuration:** during the addressed stage the host will ask the device for different information. Although the devices normally have just one configuration (Windows sets first configuration as default, this has encouraged device producers just to implement one configuration) it may be that it has more than one. When the host feels comfortable with the information it has about the device it will load the most
suitable device driver. This driver will also request information to the device and then it will set the most suitable configuration.

From this point on we can communicate with the device (in our case the keyboard) through an address and an endpoint. The specific endpoint to be used is different for every device, during the addressed stage, when the driver is gathering information, it will store the necessary endpoints.

With the information explained so far we could be able to design a general USB driver able to prepare a device for it’s specific purpose. On the following section we are going to talk about the device driver.

![Diagram](image)

Figure 7.8: Structure of the project (1). Red used to represent software. Purple used to represent hardware.

### 7.4 Specific device driver

We have learned in the previous section that when the general driver is the one in charge of discovering a new device and addressing it. When it finds a suitable driver for the device this is loaded. In this section we are going to explain the basic parts a specific driver for a device need to implement.

Adding the specific driver to our project structure will modify it by adding
a new layer. Figure 7.9 illustrates the new structure.

The functionality a specific driver needs to implement can be divided into two sets:

- General functions.
- Specific functions.

To give an example of specific functions, as we will be working with a keyboard we will use this as an example.

### 7.4.1 General functions

General functions will be implemented in all specific drivers. The set of general functions are:

- **Gather information:** as the general USB driver the specific driver should have implemented all the standard requests to get all the information needed from the device before configuring it. The number of interfaces, endpoints and configurations must be known in order to select the appropriate configuration.

- **Select configuration:** once all the information needed is known by the driver this has to select one configuration.
7.4.2 Specific functions

Depending on the device we are building the driver for specific functions will be different. For our case, a keyboard, the specific function needed will be:

- **Polling data**: polling the device to check if there is data incoming. In our specific case we want to check if a certain key is being pressed.

7.5 HUB driver

The number of ports available in a computer is rather limited. For this reason on normal USB driver implementations there is a software HUB implementation. This will allow us to connect multiple devices to use a single port and section will explain the basic functionality this driver needs to cover:

- **Data structures**: all necessary data structures for one device should
be present in the implementation. These data structures should contain all the data we have explained in the previous sections when we gathered information of the device.

- **Topology**: the hub will contain the information present on the different devices connected. The USB specification states that a hub may be connected as a device, expanding the network. In order to reach easily a certain device the hub must know where to find it.

- **Hub attach**: attach a certain device to the hub. Among other things the hub must know which are the addresses used and which are available.

- **Hub check new device**: recursively check if there is a new device attached to the hub.

- **Hub reset port**: reset a certain port exclusively.

- **Get port status**: given a port inside the hub detect the status of this.

- **Power on**: initialize data structures, recursively check new devices and initialize those.

- **Reset**: in case failure is detected it is important to reset the hub to a known state.

Our project purpose is to detect one device and communicate with it (keyboard). This is why the driver should not be entirely necessary. The basic functionality needed will be implement together with the general USB driver.

The new project structure is illustrated in figure 7.10.
7.6 USB controller

So far we have explained the message protocol between the device and the host, the necessary implementations of the specific device drivers and an overview of the necessary implementations for a hub driver. All of this would not be useful if we could not reach physically the device.

For this reason in this section we will explain the USB controller. There are two different hardware implementations present now a days in computers for the USB controller:

- Dedicated interface controller on a peripheral bus such as PCI.
- Interface controller integrated into a chip set component.

Either one implementation or the other all we have explained so far is valid and will be used to send messages to the controller, which will deliver them to the correspondent device.
Depending on the specification the standard controller may be:

- **OHCI and UHCI**: open and universal host controller interface. Both are standards for USB 1.1.

- **EHCI**: Enhanced host controller interface. Standard controller for USB 2.0.

- **XHCI**: eXtensible host controller interface. Standard controller for USB 3.0.

For this project, as we are using USB 2.0 we will also be using a EHCI controller.

The USB controller is in charge of several ports in which the devices will be connected. Figure 7.11 illustrates how the structure of the project is updated once again.

The hardware part of the structure will not change for the rest of the project. What we have as USB controller is really a EHCI controller (USB 2.0 controller). If a device not supporting USB 2.0 is connected the EHCI controller will delegate it to a companion controller.

The companion controller will configure the device using its own drivers and when the device detach the port management will return to the EHCI controller.

As we have seen with the general USB driver and the specific USB driver each USB controller has general and specific parts. The general will be the same for all the USB controllers but the specific may vary depending on the manufacturer.
In order to abstract our general USB driver from the USB controller under it we will add a new layer in between. This will be the USB controller driver.

### 7.6.1 USB controller driver

This section purpose is to explain how the USB controller driver communicates with the USB controller. Adding this new layer modifies the project structure (see Figure 7.12).
Specific USB Driver

General USB Driver

USB controller driver

USB Controller

Port

Device (i.e. Keyboard)

Figure 7.12: Structure of the project (5). Red used to represent software. Purple used to represent hardware.
USB controller registers

The USB Controller Driver will be in charge of communicating directly with
the USB controller using memory mapped I/O (MMIO) USB registers. We
can divide the registers into two subgroups:

BAR+CapabilityReg.length  Operational registers

BAR  Capability registers

Figure 7.13: Set of registers base adress

- **Capability registers**: read-only registers implemented to determine
capabilities of the host controller. For the purpose of this project these
registers will be used to gather necessary information before the actual
implementation. This will allow us to skip some steps hard-coding
the controller information required. Figure 7.13 illustrates capability
registers starting address: base address register (BAR). Also a structure
of the different registers contained is specified in the Table 7.7.

- **Operation registers**: read-write registers implemented so that the
software has a generic way of managing the host controller functionality.
The base address of these registers, as Figure 7.13 illustrates is BAR +
the length of the capability registers. These are configured among the
BIOS and the operating system (OS). In this section we will specify
which are the ones the BIOS does not set up and the ones our driver
should configure. Table 7.8 shows the whole set of operation registers.
We are just going to use port 0 for the purpose of this project. This is
the reason why in table 7.8 we just put the base address of this port
(PORTSC). If other ports status control need to be accessed they are situated in adjacent positions to port 0.

Software must read/write these registers using DWORD access only.

<table>
<thead>
<tr>
<th>Base Offset</th>
<th>Size</th>
<th>Mnemonic</th>
<th>Register name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>1 Byte</td>
<td>CAPLENGTH</td>
<td>Capability Length</td>
</tr>
<tr>
<td>01h</td>
<td>N/A</td>
<td>Reserved</td>
<td>N/A</td>
</tr>
<tr>
<td>02h</td>
<td>2 Bytes</td>
<td>HCIVERSION</td>
<td>Version number</td>
</tr>
<tr>
<td>04h</td>
<td>4 Bytes</td>
<td>HCSPARAMS</td>
<td>Structural parameters</td>
</tr>
<tr>
<td>08h</td>
<td>4 Bytes</td>
<td>HCCPARAMS</td>
<td>Capability parameters</td>
</tr>
<tr>
<td>0Ch</td>
<td>4 Bytes</td>
<td>HCSPPROUTE</td>
<td>Companion port routing</td>
</tr>
</tbody>
</table>

Table 7.7: *Capability register set*

<table>
<thead>
<tr>
<th>Base Offset</th>
<th>Size</th>
<th>Mnemonic</th>
<th>Register name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>4 Bytes</td>
<td>USBCMD</td>
<td>USB Command</td>
</tr>
<tr>
<td>04h</td>
<td>4 Bytes</td>
<td>USBSTSS</td>
<td>USB Status</td>
</tr>
<tr>
<td>08h</td>
<td>4 Bytes</td>
<td>USBINTR</td>
<td>USB Interrupt enable</td>
</tr>
<tr>
<td>0Ch</td>
<td>4 Bytes</td>
<td>FRINDEX</td>
<td>USB Frame index</td>
</tr>
<tr>
<td>10h</td>
<td>4 Bytes</td>
<td>CTRLDSSEGMENT</td>
<td>4G Segment selector</td>
</tr>
<tr>
<td>14h</td>
<td>4 Bytes</td>
<td>PERIODICLISTBASE</td>
<td>Frame list base address</td>
</tr>
<tr>
<td>18h</td>
<td>4 Bytes</td>
<td>ASYNCLISTADDR</td>
<td>Next asynch list address</td>
</tr>
<tr>
<td>1Ch - 3Fh</td>
<td>N/A</td>
<td>Reserved</td>
<td>N/A</td>
</tr>
<tr>
<td>40h</td>
<td>4 Bytes</td>
<td>CONFIGFLAG</td>
<td>Configured flag register</td>
</tr>
<tr>
<td>44h</td>
<td>4 Bytes</td>
<td>PORTSC</td>
<td>Port 0 status control</td>
</tr>
</tbody>
</table>

Table 7.8: *Operational register set*
USB controller during BIOS initialization

During boot the BIOS will initialise a set of PCI configuration parameters necessary to set up the USB controller. We will not enter into much detail about this part because it is supposed to be done by the time our driver enters into action. Although these are some of the relevant parts done:

- Write and enable the base address register (BAR) for accessing the structural parameters registers.
- Write the subsystem vendor information into the capability registers.
- Write the number of ports available.
- Write the default parameters on ports in case the OS does not select a specific one.

Ownership of the controller

Once the BIOS is done setting up the OS starts loading. The first thing our controller driver should do is to claim ownership of the controller. Exclusivity control over the USB controller is necessary for our driver. The steps are (see Figure 7.14):

1. Set HC OS owned semaphore = 1
2. Wait until HC BIOS owned semaphore = 0

Once we gain the ownership over the controller the BIOS is not allowed to claim back the ownership (while the computer is on), for this reason we won’t need to implement any change in ownership detection strategies.
Return the controller to a default state

As we do with devices when plugged in, we need to reset the controller to return it to a deterministic (default) state. This will make all operational registers, including port registers and port stat machines return to their initial values.

To initiate a host controller reset system software must:

1. **Stop the host controller:** to do so the driver must set to 0 the USB2CMD.Run/Stop memory position.

2. **Wait until the host has halted:** the driver should check the value of the memory position USB2STS.HCHalted. This position will have a 1 as soon as the controller is halted.
3. **Begin the reset:** once the controller is halted, never before, a 1 must be written to the memory position USB2CMD.HostControllerReset. This will make the controller to begin the reset.

4. **Wait for the reset:** the driver would wait until the controller finishes to reset. When it is done it will write a 0 to the memory position USB2CMD.HostControllerReset.

### Configure the controller

Once the reset is complete the controller returns to a default state. Is now moment to initialize the different parameters necessary for the controller to work properly.

The first thing we will set is the CTRLDSSEGMENT register. This register sets the 32 most significant address bits used for all the controller data structures. If the operating system we are working with is a 32 bit OS we will set this register to all zeros.

The next configuration step may be decisive depending on the type of USB driver we want to implement. We have two possible implementations:

- **Interrupts:** we may want to set software interrupts for certain events. These events will trigger a function to determine what has caused the interruption.

- **Polling:** there may be the case we prefer to perform a polling method to detect certain events. The same events that produce an interruption may be configured to be checked through polling.
There are six different events to configure. These methods are not exclusive, we may want to configure just one event with interruption and the other five with polling. The events are represented in the USBINTR register:

- Async advance enable (bit 5).
- Host system error (bit 4).
- Frame list rollover (bit 3).
- Port change (bit 2).
- USB error (bit 1).
- USB interrupt enable (bit 0).

For more information of these events the USB 2.0 specification may be checked. For our project we will just use the port change either we use polling or interrupts.

In case we decide to set some events with interrupts we must set to one the specific bit of the USBINTR register. When we receive a software interruption we may then check the register USBSTSS to know which is the event that triggered the interruption.

If we have decided to set some interrupts we must program the desired interrupt threshold via the register USBCMD.InterruptThresholdControl.

When the events are configured is time for the controller to start running. We must set the USBCMD.Run/Stop bit to one.

Once the above steps have been completed the port registers will begin reporting device connects, disconnects, etc.
7.7 Detecting a device plug in

This section will be using the PORTSC register from the operation register set. For a better comprehension of the PORTSC Table 7.9 illustrates the division inside this register.

When the controller detects a new attachment, it will set the port change bit in the PORTSC register of that port (port 0 in our case). The controller will also set the current connect status bit in the same register. If interrupts were enabled the host controller will generate a port-change interrupt at the next interrupt threshold.

The USB driver, upon receipt of the interrupt, will identify the port checking the port change bit in the PORTSC register. Then the driver will use the GetPortStatus() request to check the nature of the change. It will then detect the change is due the connect status change bit. This bit must be cleared once is read.

Once we know the port the new device is attached we must perform a reset to the port, wait the reset is done and enable the port.

The port will be now functional and the device is in powered state.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:23</td>
<td>Reserved</td>
</tr>
<tr>
<td>22</td>
<td>Wake on over-current enabled (R/W).</td>
</tr>
<tr>
<td>21</td>
<td>Wake on disconnect Enable (R/W).</td>
</tr>
<tr>
<td>20</td>
<td>Wake on connect enable (R/W).</td>
</tr>
<tr>
<td>19:16</td>
<td>Port test control (R/W)</td>
</tr>
<tr>
<td>15:14</td>
<td>Port indicator control (R/W)</td>
</tr>
<tr>
<td>13</td>
<td>Port owner (R/W)</td>
</tr>
<tr>
<td>12</td>
<td>Port power -PP (R/W)</td>
</tr>
<tr>
<td>11:10</td>
<td>Line status (RO)</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Port reset(R/W)</td>
</tr>
<tr>
<td>7</td>
<td>Suspend (R/W)</td>
</tr>
<tr>
<td>6</td>
<td>Force port resume (R/W)</td>
</tr>
<tr>
<td>5</td>
<td>Over-current change (R/W)</td>
</tr>
<tr>
<td>4</td>
<td>Over-current active (RO)</td>
</tr>
<tr>
<td>3</td>
<td>Port enable/disable change (R/W)</td>
</tr>
<tr>
<td>2</td>
<td>Port enable/disable (R/W)</td>
</tr>
<tr>
<td>1</td>
<td>Connect status change (R/W)</td>
</tr>
<tr>
<td>0</td>
<td>Current connect status (RO)</td>
</tr>
</tbody>
</table>

Table 7.9: PORTSC bits description
Chapter 8

Implementation’s design

The purpose of this chapter is to describe the design to implement a driver capable of communicating a keyboard with an OS. We start detailing an overview of the whole implementation to give the reader an abstract view of what we have designed. Then the chapter is divided into the different steps (ordered chronologically) we must perform to finally communicate a device with the OS. Given the complexity of the USB protocol we have decided not to specify all the data structures that could distract the reader from the protocol itself, for more information we recommend reading the USB 2.0 specifications.

8.1 Design overview

This section will recap the different steps needed to go through (in an abstract way) to implement the USB driver.
Without entering into much detail these are the eight steps that we will need to follow:

1. Initialize EHCI controller driver.
2. Gain ownership of the EHCI controller
3. Initialize EHCI controller
4. Initialize USB general driver and Hub driver
5. Scan port until we detect a new device plugged.
6. Initialize device.
7. Load suitable driver (Keyboard driver).
8. Configure device.

To do so we need to implement five different components:

- **EHCI driver.** The bottom layer of our structure. It will communicate directly with the EHCI controller. The implementation must contain:
  - Set up structures for the operational registers.
  - Gain ownership of the controller
  - Configure the controller

- **USB core driver.** This will be our main driver. Three basic things must be implemented:
  - Attachment of a new device (plug in).
• **USB basic data structures.**

  • **Transfer class.** This class will contain the implementation of packets, transactions and transfers. It will be a helper class for the USB core driver.

  • **HUB class.** Helper class for USB core driver. Among other things it must implement:
    - Root hub and hub functionality.
    - Scan port.
    - Attach new device.

  • **USB device driver.** Specific driver for the keyboard. It will contain the functions necessary for the keyboard to work:
    - Check if we have a keyboard
    - Configuring the device.

**8.2 Initialize EHCI controller driver**

Our basic work in this section will be to set up the data structures of the capability and operational registers.

The base address register (BAR) will need to be provided. With the BAR address and help of the Tables 7.7 and 7.8 we will be able save the base address for the registers we will use.
The pseudo-code to store the base address needed is:

```plaintext
set(operationalBase, BAR+CapabilityRegisters.length);
set(USBCMD, operationalBase);
set(USBSTSS, operationalBase + 04h);
set(USBINTR, operationalBase + 08h);
set(CTRLDSSEGMENT, operationalBase + 10h);
set(PORTSC, operationalBase + 44h);
```

It is important to take in mind software must read/write these registers using DWORD access only.

### 8.3 Gain ownership of the EHCI controller

Gaining the ownership of the EHCI controller will be done using semaphores.

The basic pseudo-code used for this minimal version will be the following:

```plaintext
set(HC OS Owned semaphore, 1);
while HC BIOS Owned semaphore == 1 do
    sleep(1s);
end
```

### 8.4 Initialize EHCI controller

During the initialization of the EHCI controller two steps need to be done:

1. Reset the EHCI controller to return it to a deterministic default stage.
2. Basic configuration of the controller.

8.4.1 Reset the EHCI controller

The base pseudo-code to reset the EHCI controller is:

```plaintext
setOperationRegister(USBCMD.Run/Stop, 0);
while readOperationalRegister(USBSTSS.HCHalted) == 0 do
  sleep(1s);
end
setOperationalRegister(USBCMD.HostControllerReset, 1);
while readOperationalRegister(USBCMD.HostControllerReset) == 1 do
  sleep(1s);
end
```
8.4.2 Basic configuration of the controller

Now that we know the controller is in default stage and all the registers are back to default we need to configure some registers:

/*Address should contain the 32 most significant bits used for all the controller data structures.*/
setOperationalRegister(CTRLDSSEGMENT, Address);
/*We don’t want interruptions, we will use polling for this exercise so we set USBINTR to 0 for all events. */
setOperationalRegister(USBINTR, 0);
/*Turn on the the controller*/
setOperationalRegister(USBCMD.Run/Stop, 1);

8.5 Initialize the general USB driver

The initialization of the general USB driver is split in three parts:

- Set up the PORTSC structure
- Set up the transfer class
- Set up the Hub class
8.5.1 Set up the PORTSC structure

The PORTSC structure will be used to communicate with a certain port. The following pseudo-code makes reference to the Table 7.9:

```
set(currentConnectState, 0b);
set(connectStatusChange, 1b);
set(portEnableDisable, 2b);
set(portEnableDisableChange, 3b);
set(overCurrentActive, 4b);
set(overCurrentChange, 5b);
set(forcePortResume, 6b);
set(suspend, 7b);
set(portReset, 8b);
set(lineStatus, 10b);
set(portPowerPP, 12b);
set(portOwner, 13b);
set(portIndicatorControl, 14b);
set(portTestControl, 16b);
set(wakeOnConnectEnable, 20b);
set(wakeOnDisconnectEnable, 21b);
set(wakeonOverCurrentEnable, 22b);
```

Using the EHCI controller driver we will be able to access the PORTSC. Then we will use these registers to set up the different ports.
8.5.2 Set up the transfer class

This class will be used by the main driver to send and receive packets in an abstract way.

The basic packet structure illustrated in Figure 7.3 will be the core of this class. Using this we will codify the PID of the four types of packets and its subtypes (check Tables 7.1, 7.2, 7.3 and 7.4 for more information).

these packets will be used in the IN, OUT and Setup transactions. The basic structure of these transactions has been defined in the section 7.2.2 of this project.

For last, the transactions will be encapsulated to built up the four types of transfers available in USB, these transfers are the Control, Bulk, Interrupt and Isochronous transfers (explained in detail in section 7.2.3).

Thanks to this transfer class we will be able to send and receive messages without having to codify them every time. It will suppose an abstraction level for our general USB driver.

8.5.3 Set up the Hub class

The software Hub implementation, although we will just have one device, will need to implement a roothub where to attach a device.

The structure in our case will just be of a roothub attached to a port with a device as a leaf (structure illustrated in Figure 8.1).

The basic structures necessary for this implementation are well defined in
the USB 2.0 specification:

- Hub descriptor is defined in section 11.23.2.1.
- Hub status structure is defined in section 11.24.2.6.
- Hub status change is defined in section 11.24.2.6.
- Hub port status is defined in section 11.24.2.7.1.
- Hub port status change is defined in section 11.24.2.7.2.

And the necessary functions we will need to implement are:

- **Hub port reset**: reset a port on the hub (root hub in our case).
- **Hub check**: check if there is a new device connected to the hub.
- **Hub attach**: attach a device to the hub.
8.6 Scan port until we detect a new device plugged

To detect if a device is plugged in we may have activated the interruptions or the polling. To make this demonstration simpler we decided to use the polling method. The following pseudo-code shows the basic implementation:

[start] Wait until a port change is detected. ;
while readOperationalRegister(PORTSC.PortChangeDetect) == 0 do
    sleep(1s);
end

We have found a change detected in the port. We now need to clear the bit. ;
setOperationalRegister(PORTSC.PortChangeDetect, 0);

Check if the port change detect is from a device plugged in, if not return to [start];
if PORTSC.CurrentConnectStatus == 1 then
    We have a device connected ;
    setOperationalRegister(PORTSC.ConnectStatusChange, 1);
    Indicate that the port is enabled ;
    setOperationalRegister(PORTSC.PortEnabled/Disabled, 1);
    Indicate that the port is activated ;
    setOperationalRegister(PORTSC.Suspend, 0);
else
    go to [start]
end
8.7 Initialize device

Once the port is activated we can initialize the device. The first thing we need to do is perform a reset to the device to return it to a deterministic default stage. Before that let’s create the reset function:

```c
setOperationalRegister(PORTSC.Suspend, 1);
sleep(1s); setOperationalRegister(PORTSC.Reset, 1);
sleep(1s);
```

With the reset function we can initialize the device:

```c
resetDevice();
controlTransfer(getDescriptor(device));
resetDevice();
controlTransfer(setAddress(newAddress, device));
sleep(1s);
controlTransfer(getDescriptor(device));
controlTransfer(getInterfaces(device));
controlTransfer(getStatus(device));
```

The minimal version of the above pseudo-code for a keyboard device should be:

```c
resetDevice();
controlTransfer(setAddress(newAddress, device));
sleep(1s);
```
8.8 Load suitable driver (Keyboard driver)

At this point the device (keyboard) is in addressed stage. The specific device driver will load and will gather information about the device. This is done to ensure the correct driver has been loaded and to choose the appropriate configuration.

The pseudo-code for this part is an extract form the *initialize the device*.

```c
controlTransfer(getDescritpor(device));
controlTransfer(getInterfaces(device));
controlTransfer(getConfiguration(device));
```

8.9 Configure device

Last section before we can use our device for the purpose it was meant to. The only thing we have to do is set the most appropriate configuration (if more than one provided).

```c
controlTransfer(setConfiguration(configuration, device));
```

Now we can communicate with the device using the endpoints specified in the configuration chosen.
Chapter 9

Economic valuation

This chapter purpose is to do an economic valuation of the project. To do so we will present a comparison between the time and the costs specified in the project planning and the real ones. Also, an introduction to the future work necessary to continue with this project will be presented.

9.1 Time distribution among tasks comparison

During the Chapter 4 Project planning we estimated the time each task should had lasted. Table 9.1 contains the comparison between the estimated and the real time.

When we first started this project we had the experience of working with serial keyboards. The communication with this hardware normally consists
of a set of memory address to communicate directly with the controller. We
believed we could do something similar with the USB controller and the focus
of the project would be just understanding the communication protocol.

As we explained in chapter 7: USB reverse engineering the communication
protocol was just a little part of all we had to learn. Instead of what we
initially though we ended up understanding, explaining and designing:

- Device driver.
- General usb driver.
- Hub class.
- Transfer class.
- USB controller driver.

We could have included third-party libraries to cover the unexpected layers
but, we thought it would be much more interesting to research, explain and
design all the different layers involved.

For this reason all the time initially assigned to the implementation set of
tasks is finally designated to the approach and design.
<table>
<thead>
<tr>
<th>Task</th>
<th>Estimated hours</th>
<th>Real hours</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total GEP</td>
<td>70</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>Deliverable 1</td>
<td>25</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Deliverable 2</td>
<td>9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Deliverable 3</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Deliverable 4</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Deliverable 5</td>
<td>19</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td><strong>Approach to USB protocol</strong></td>
<td></td>
<td></td>
<td>+85</td>
</tr>
<tr>
<td>Research of USB</td>
<td>10</td>
<td>25</td>
<td>+15</td>
</tr>
<tr>
<td>Study of the protocol</td>
<td>30</td>
<td>110</td>
<td>+70</td>
</tr>
<tr>
<td><strong>Design of the USB driver</strong></td>
<td></td>
<td></td>
<td>+65</td>
</tr>
<tr>
<td>Pairing the device</td>
<td>40</td>
<td>105</td>
<td>+65</td>
</tr>
<tr>
<td>Pulling and retrieving data</td>
<td>20</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>150</td>
<td>0</td>
<td>-150</td>
</tr>
<tr>
<td>Implementing the part</td>
<td>70</td>
<td>0</td>
<td>-70</td>
</tr>
<tr>
<td><strong>Pairing to a device</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementing the part</td>
<td>50</td>
<td>0</td>
<td>-50</td>
</tr>
<tr>
<td><strong>Pulling and retrieving data</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>30</td>
<td>0</td>
<td>-30</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td>30</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>350</td>
<td>350</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.1: Time distribution among tasks
9.2 Real budget distribution

On the previous section we compared the time distribution among tasks of the project planning with the real one. We also justified why this time distribution was different of what we initially thought.

Table 9.2 illustrates the real budget distribution among the set of tasks given the new time distribution.

<table>
<thead>
<tr>
<th>Set of tasks</th>
<th>Estimated budget</th>
<th>Real budget</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEP</td>
<td>2.859€</td>
<td>2.859€</td>
<td>0€</td>
</tr>
<tr>
<td>Approach to USB protocol</td>
<td>1.586€</td>
<td>5.612€</td>
<td>+4.026€</td>
</tr>
<tr>
<td>Design of the USB driver</td>
<td>2.576€</td>
<td>5.797€</td>
<td>+3.221€</td>
</tr>
<tr>
<td>Implementation</td>
<td>7.247€</td>
<td>0€</td>
<td>-7.247€</td>
</tr>
<tr>
<td>Total</td>
<td>14.270€</td>
<td>14.270€</td>
<td>0€</td>
</tr>
</tbody>
</table>

Table 9.2: Budget distribution among set of tasks.

9.3 Future work

Once all the different layers involved in the plug in of a device have been described and designed the last pending thing to do is the implementation.

The precise description of the different layers will help future students to continue with this work. Also, with the understanding of the complexity the USB protocol has they will be able to plan the time distribution much more precisely.
Chapter 10

Conclusions

The main ideas behind this project where:

- Perform a reverse engineering of the USB protocol in order to describe and document in a comprehensive way how the protocol works, and a precise dissection of it.

- Step by step implementation example of a driver capable of enabling the communication between a specific keyboard with a certain operating system.

As we explained in the chapter 7 USB reverse engineering the universal serial bus was meant to satisfy the plug-and-play requirement. To do so the USB driver must be able to initialise the controller, implement a software hub, detect when a device is plugged in and load a suitable driver for this specific device.

What we thought it will be a unique layer to implement the USB driver

81
transformed rapidly to several layers, each one containing its peculiarities. During this project we have not just explained these different layers but designed what the implementation should look like.

Separating the different set of layers will allow us to implement future devices much easier, as the common part will be the same. The only part we will need to re-design each time is the specific device driver.

As the amount of layers was different from what we thought, the time to research, understand and design the whole set was much bigger than what we initially planned. Unfortunately, this time deviation left us no time to implement what we designed.
Bibliography

   http://www.usb.org/home

   http://libusb.info/


[4] CentOS Operating System. More informations about it can be found in the official web page:
   https://www.centos.org/

[5] Site with multiple projects related with computer science. One is about USB. This page was visited on September 2016.
   http://wiki.beyondlogic.org

   https://github.com/Chadderz121/csud
[7] Site explaining the SCRUM-Agile methodology. This page was visited on September 2016.

http://scrummethodology.com/


https://trello.com/


https://github.com/


http://upcommons.upc.edu/handle/2117/78899

[11] USB 2.0 specification

http://www.usb.org/developers/docs/usb20_docs/usb_20_120516.zip

[12] USB made simple web page

http://www.usbmadesimple.co.uk