HID interface program based in Arduino for connecting flying simulator controls to a PC

-

Programa interfaz HID basado en Arduino para conectar controles de simulador de vuelo a PC

GRADO EN INGENIERÍA INFORMÁTICA DEL SOFTWARE

TRABAJO DE FIN DE GRADO

AUTOR

Pablo Suárez García

TUTOR

Miguel Ángel González Muñoz

Julio 2022

Este documento ha sido creado basándose en la plantilla elaborada por JOSÉ MANUEL REDONDO LÓPEZ. [1] [2]

Table of contents

Figures in the document

Tables in the document

Chapter 1 - About this paper

Chapter 1.1 - Abstract

The work seeks the elaboration of a software for Arduino boards that transform them in a unit able to translate physical interaction from electronics such as switches and potentiometers into controls that can be interpreted by a computer though the Human Interface Devices protocol by the USB Implementers' Forum. In this paper decisions about which Arduino boards would be adequate for the purpose with their advantages are documented. Explanation about the software development is included serving as a documentation for both what has been done, including specifics in challenges found and their solutions, and how it could be replicated to the point that anyone with some basic knowledge in Arduino programming could adapt it to their specific needs being those a different board or set of controls. And finally the design and assembly of a working protype featuring an Arduino Leonardo board has been carried in order to both serve as testing base for the developed software and to demonstrate that the task can indeed be carried by anyone with a minimum base in electronics and crafting with materials accessible through common means without inquiring into customized hardware that may only be acquired by big assemblers.

Chapter 1.1.1 - Resumen

El trabajo busca la elaboración de un software para placas Arduino que las transforme en una unidad capaz de traducir la interacción física de componentes electrónicos como interruptores y potenciómetros en controles que puedan ser interpretados por un ordenador a través del protocolo Human Interface Devices del USB Implementers' Forum. En este documento se recogen las decisiones sobre qué placas Arduino serían adecuadas para el propósito con sus ventajas. Se incluye una explicación sobre el desarrollo del software que sirve como documentación tanto de lo que se ha hecho, incluidos los detalles de los desafíos encontrados y sus soluciones, y de cómo podría replicarse hasta el punto de que cualquier persona con algunos conocimientos básicos en programación Arduino podría adaptarlo a sus necesidades, sean estas el uso de otra placa u otro conjunto de controles. Finalmente, se ha llevado a cabo el diseño y montaje de un prototipo con una placa Arduino Leonardo con intención de usarse como base de prueba para el software desarrollado y para demostrar que la construcción puede ser realizada por cualquier persona con una base mínima en electrónica y montaje con materiales accesibles por medios comunes sin necesidad de hardware personalizado sólo al alcance de grandes ensambladores.

Chapter 1.2 - Keywords

Arduino, HID, Controller, Gamepad, Joystick, Electronics, coding.

Chapter 1.2.1 - Palabras clave

Arduino, HID, Controlador, Mando de juegos, Joystick, Electrónica, programación.

Chapter 2 - Requirements and objectives

We are first to analyse the project title and after debate determine what is expected to be performed in the development of the project. Then additionally specify some extra objectives that will result in desirable characteristics that would add value to any product result of the development or compose some knowledge with a value on its own.

The starting point for the objectives is the title in Spanish for the project, “Programa interfaz HID basado en Arduino para conectar controles de simulador de vuelo a PC”, which translated into English is “HID interface program based in Arduino for connecting flying simulator controls to a PC”. No further description is provided at the beginning.

Chapter 2.1 - Base project requirements

The only established at first objective is the development of a software for an Arduino board that allows to use said board as an interface for flying simulator controls in a host system. This objective can be broken down into developing a software accomplishing the following:

- The board will use the HID protocol to identify itself.
- The board has to communicate the controls it will report
- Controls have to be reported.
- Implicit to the use of HID protocol comes that the board will be recognized and used by the host system without installing any proprietary driver.

These objectives are deemed only as a description to a concept of what wants to be developed, but while not implicit some qualities are desirable for the device that can be inferred from that concept. For this matter after some debate further objectives were established for the project.

Chapter 2.2 - Established project objectives

It is determined that the final scope of the project shall not only include a software development, but that an investigation in the Arduino ecosystem has to be carried to see the viability of the task at hand and determine the best way it can be carried while also documenting the obtained knowledge. It is also deemed necessary to demonstrate not only necessary to develop the software but building a prototype that demonstrates that the device defined by the reached concept is possible and can indeed work in the real world. Apart from those requirements established as a base in the previous chapter we define the following for both the software and the protype.

Chapter 2.2.1 - Objectives for the developed software solution

- This software is expected to work on at least one commonly available Arduino board. Investigation about other capable boards is expected.
    - It should be able to translate controls of the following kinds:
    - Button.
    - DPAD/Hat switch.
- Axes, maximum accuracy possible.
- The number of controls that may be implemented depending on the board should be considered and an adequate balance between them determined.
- The way the control interpretation is done should allow for the simplest wiring of the electronic part of a resulting device.
- The controller should be initialized with no further interaction than connecting the board to the host system.
- A way of switching between accuracy level for the axes in order to prevent noise with basic potentiometers.

- The control interpretation has to be performed in the most generic way possible, not expecting specific electronic elements that may have different behaviours.

Chapter 2.2.2 - Testing protype objectives

- The prototype is to be based on a commonly available Arduino board.
- The prototype is to make use of the connectivity capabilities of the selected board.
- It has to be connected to a host system though an USB connection not requiring additional power supply.
- It is to be used as a way to demonstrate the simplest way to wire each control.
- It has to allow for fast and preferably tool less modification to allow the testing of different characteristics.
- It must present physically every control included in the software.
- It has to present a way to switch between the accuracy modes stipulated for the software.
- Its main objective is to demonstrate the possibility of assembling this kind of device and then demonstrating the functionality of the software.

Chapter 3 - Viability study and hardware decisions

In this section the initial objective of just making a working software for an Arduino board to fulfil the desired function is considered. There exists the possibility that no board available is able to carry this task in a satisfying way and as such all the mainstream boards have be analysed.

It is a clear that for a solution to be satisfactory two main requirements have to be fulfilled, that is for the final device to be able to handle an acceptable number of controls and to do that in a responsive and precise manner. And additional requirement that is relevant for the project to be both viable and of use is for it to operate in a widely available board, otherwise it may not be repeatable in the future either by my or any future user.

With all the previous statements in mind it can be said that the project will be viable if an Arduino board able to handle a decent amount of both digital and analogic controls while keeping a fast USB connection. Otherwise, the project objectives would have to be changed in order to include other microcontrollers as possible in the scope, which might defeat the purpose of using a cheap and open-source board.

Chapter 3.1 - Widely available Arduino boards and their capabilities

Chapter 3.1.1 - Uno Rev3

Information from [3] and [4].

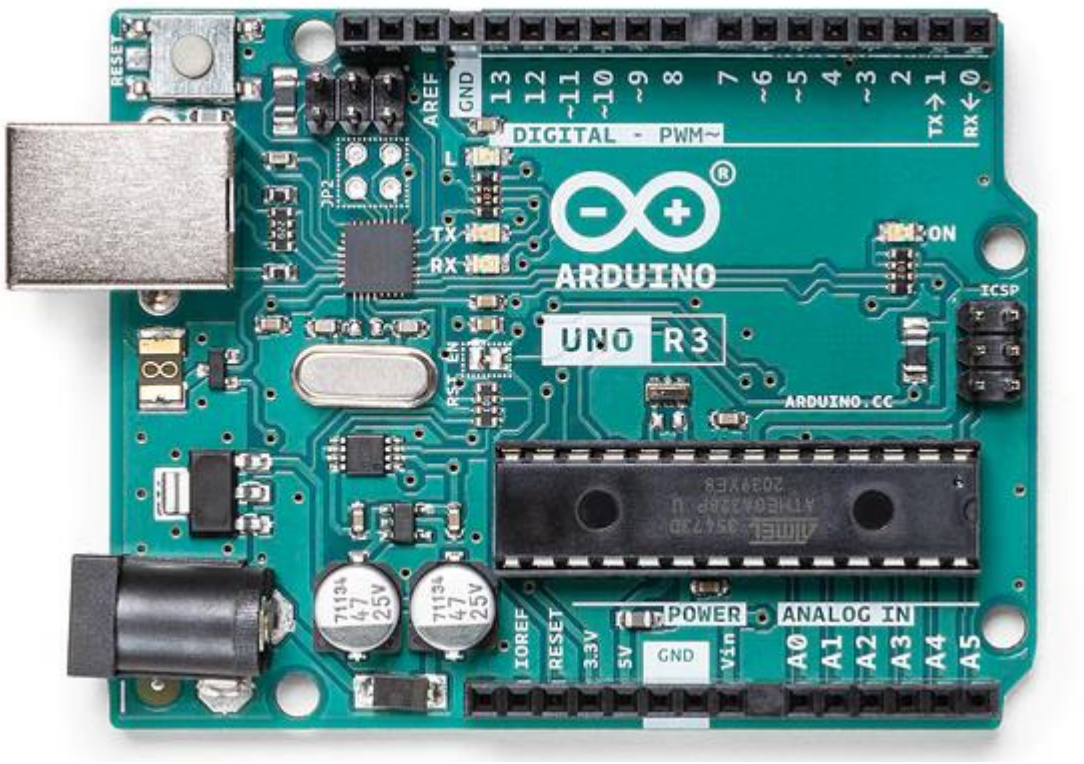

Figure 1 - Arduino UNO Rev 3

Currently running its third revision, the Uno R3 is the most basic board we can find. The original Uno was the first USB board released and it is considered the reference for Arduino boards.

Chapter 3.1.1.1 - Technical Specs

Board	**Name**	Arduino UNO R3
	SKU	A000066
Microcontroller	ATmega328P	
USB connector	USB-B	
Pins	**Built-in LED Pin**	13
	Digital I/O Pins	14
	Analog input pins	6
	PWM pins	6
Communication	**UART**	Yes
	I2C	Yes
	SPI	Yes
Power	**I/O Voltage**	5V
	Input voltage (nominal)	7-12V
	DC Current per I/O Pin	20 mA
	Power Supply Connector	Barrel Plug
Clock speed	**Main Processor**	ATmega328P 16 MHz
	USB-Serial Processor	ATmega16U2 16 MHz
Memory	**ATmega328P**	2KB SRAM, 32KB FLASH, 1KB EEPROM
Dimensions	**Weight**	25 g
	Width	53.4 mm
	Length	68.6 mm

Table 1 - Arduino Uno Rev 3 Technical Specs

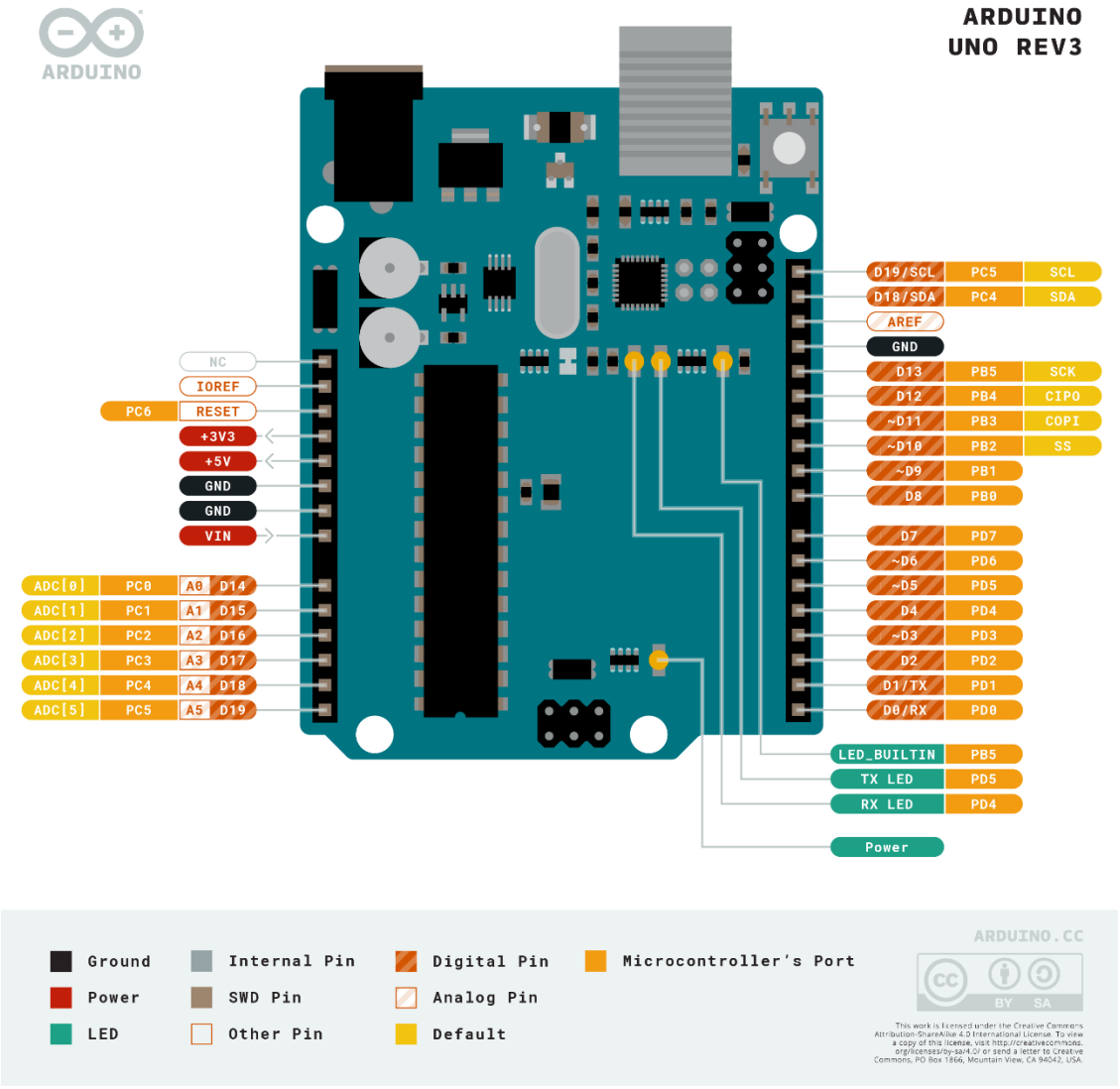

Figure 2 - Arduino Uno Rev 3 Pinout

Some notable and relevant specs about this board are the use of a main processor that will execute the program loaded into the board and it is wired to the pins and thus will carry the readings and writings, and then an extra co-processor performance tasked with managing the USB connection used for programming and sending the serial bus from the main processor.

The main processor serial bus is a TTL serial wired to the digital pins 0 and 1, these two pins may be use for either the serial bus or as digital I/O to connect extra electrical devices if the rest of the pins are already in use. The ATmega16U2 co-processor replicates this serial bus through the USB-B port that is also used for programming the board and is generally used to output the serial monitor to the Arduino IDE, but it does not operate as a regular USB device.

Some additional regards about this board are the size, standing at a footprint of 53.4 by 68.6 milimeters it is not the most adequate board to be assembled into a compact device, but for the project at hand this is not an issue due to the kind of prototype intended. The price of this board in the official store is 20€ making it quite accessible. At the moment of writing this paper a limited run for a more compact version with similar capabilities called the UNO Mini Limited Edition can be found for 40€ that could enable users to assemble a more compact device.

https://www.arduino.cc/reference/en/language/functions/communication/serial/

Chapter 3.1.1.2 - Theoretical limits for the application

In the supposition that the board could be programmed to carry the task of communicating the physical inputs to a computer as a HID device this board would handle the following:

- Up to 6 axes through the analog pins.
  - Reading precision of 10 bits, meaning 1024 possible values.
  - Reading takes 100 microseconds according to Arduino reference, resulting in an approximated max sample rate of 10000 times a second. [5]
- Up to 20 digital controls if every digital and the 6 analog pins were used.
  - Digital pins 0 and 1 could be used for serial I/O limiting the button amount, pin 13 can also be left free for the built in LED to be used.

These theoretical limits are established considering the existing pins in the board and not the real capabilities of the board that may only be found when attempting the final implementation.

Chapter 3.1.1.3 - Conclusions

This board could be an interesting option for the development for being the most absolute baseline of what an USB Arduino is, meaning if it can be done with this board it could theoretically work in any other one. Unfortunately, not having a native USB connection for the main processor is a huge limitation in the amount of data to be carried resulting in a poor pooling rate and thus driving the controls unresponsive if it even was possible to establish the device as a HID one.

Chapter 3.1.2 - Due

Information from [6] and [7].

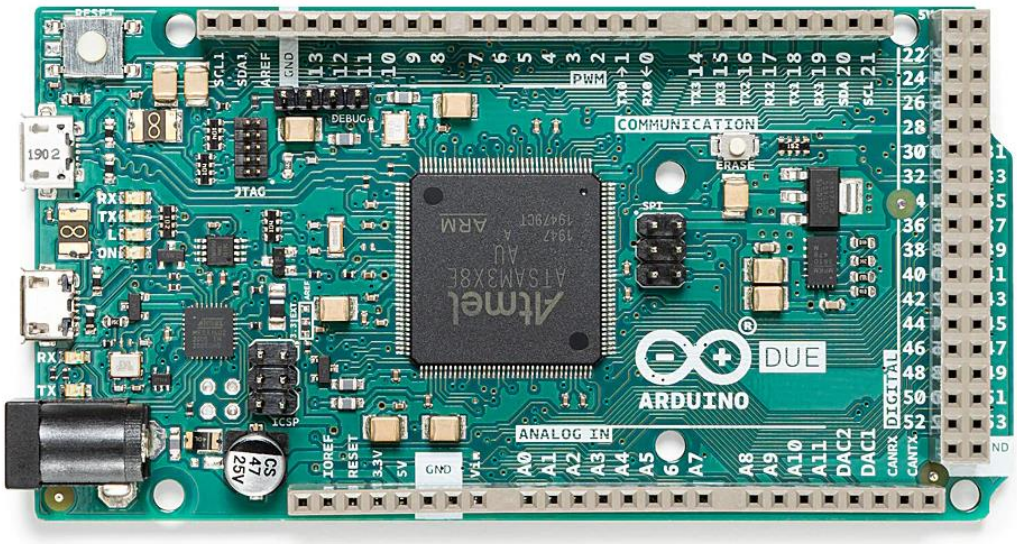

Figure 3 - Arduino Due

This board is a clear step up from the Uno in the term of capabilities, featuring a 32-bit ARM microcontroller it has ample connectivity for controls and even features a native USB port and is indicated by Arduino to be able to use the mouse and keyboard libraries, both of which rely on the HID library serving as a starting point for implementing the solution. This is a promising overview for this board to be selected, but some other characteristics make it less applicable for this specific task.

Chapter 3.1.2.1 - Technical Specs

Board	**Name**	Arduino® Due
	SKU	A000062
Microcontroller	AT91SAM3X8E	
USB connector	Micro USB	
Pins	**Built-in LED Pin**	13
	Digital I/O Pins	54
	Analog input pins	12
	Analog output pins	2
	PWM pins	12
Communication	**CAN**	Yes (ext. transceiver needed)
	UART	Yes, 4
	I2C	Yes
	SPI	Yes
Power	**I/O Voltage**	3.3V
	Input voltage (nominal)	7-12V
	DC Current per I/O pin (group 1)	9 mA
	DC Current per I/O pin (group 2)	3 mA
	Power Supply Connector	Barrel Plug
	Total DC Output Current on all I/O lines	130 mA
Clock speed	**Processor**	AT91SAM3X8E 84 MHz
Memory	**AT91SAM3X8E**	96KB SRAM, 512KB flash
Dimensions	**Weight**	36 g
	Width	53.3 mm
	Length	101.5 mm

Table 2 - Arduino Due Technical Specs

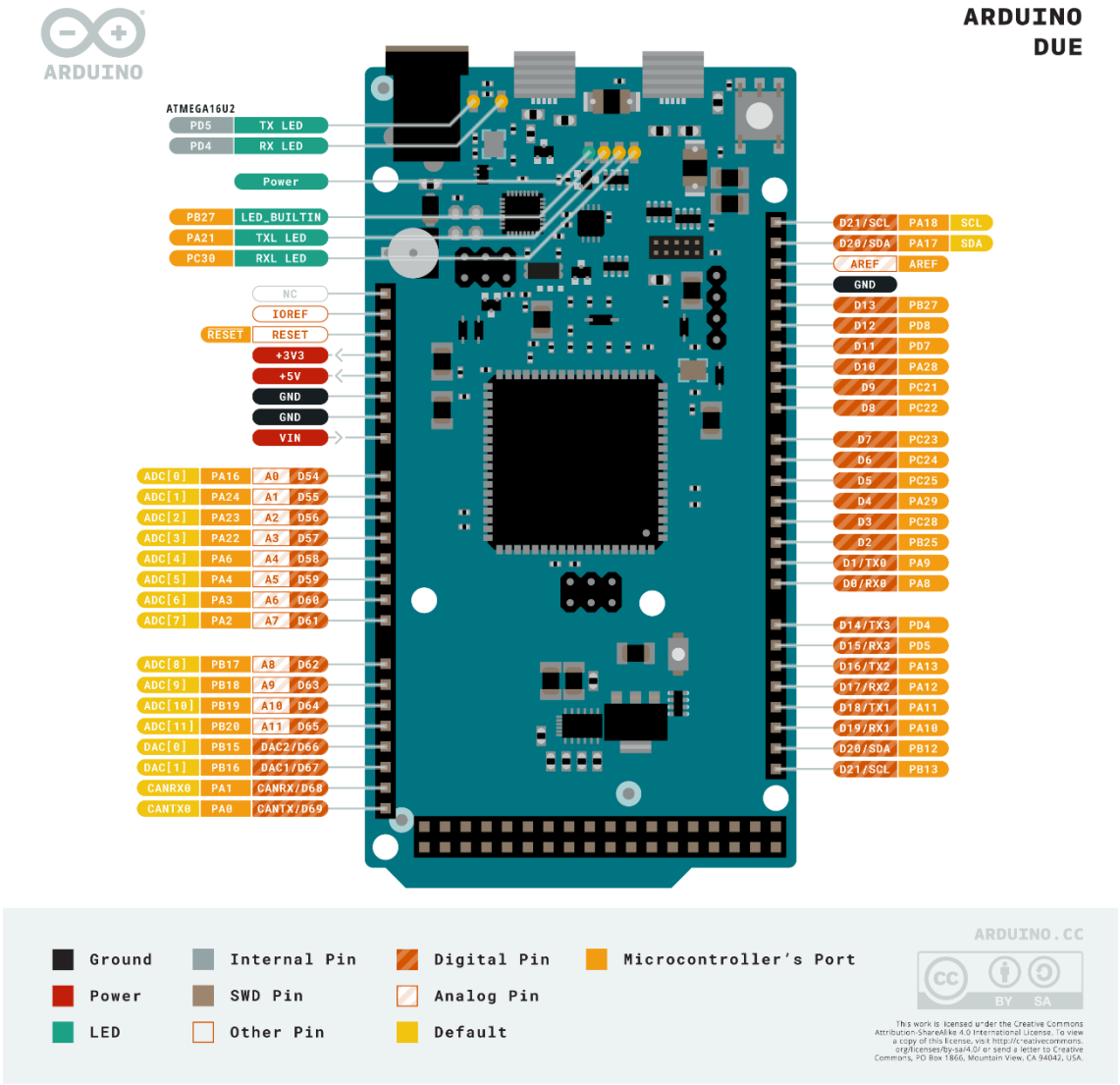

Figure 4 - Arduino Due Pinout 1

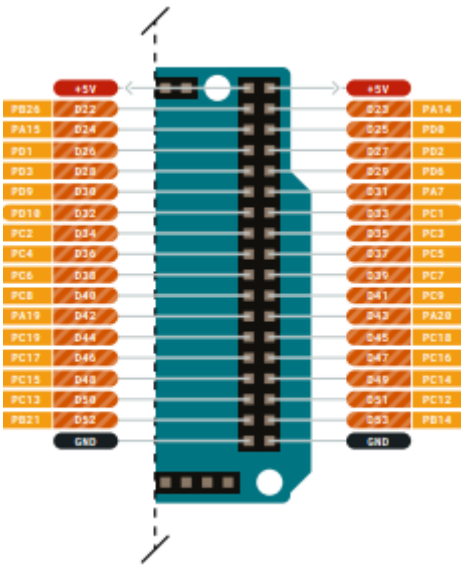

Figure 5 - Arduino Due Pinout 2

Two especially important points can be taken from the spec sheet, first, the great amount of both digital I/O and analog pins that could enable the board to manage an enormous amount of control devices. Secondly, the weakest point of this board, the limitation of voltage to 3.3 volts. When setting up analog control devices the most common is to use 5 volts, it is possible to work around this with the use of adequate electronics, but it would constitute an extra requirement when a complete controller is assembled.

The Due presents an ATmega16U2 co-processor that handles the USB connection for programming the board and the principal serial communication as in the Uno, but unlike the prior one it takes advantage in providing an additional native USB port with capability to use the Arduino core HID library that serves as basis for the implementation of the software part of this project.

The Due has a bigger than usual footprint of 53.3 by 101.5 milimeters that really poses a challenge for compact integrations, but it could still be serviceable for bigger setups such as those using structures such as tables or frames as chassis to attach the controls to. This board capabilities tax the final price that stands a 35€, making it not that suitable for economic setups but still viable for the higher end ones.

Chapter 3.1.2.2 - Theoretical limits for the application

As for the other boards, the limits regarding the existing pins are considered and not the real limits found by testing.

- Up to 12 axes through the analog pins.
  - Reading precision of 12 bits resulting in 4096 possible values.
  - Reading takes 100 microseconds according to Arduino reference, resulting in an approximated max sample rate of 10000 times a second. [5]
- Up to 66 digital controls if every digital and the 12 analog pins were used.
  - As with other boards the use of some pins would disable their use as serial I/O, in pairs, 0 and 1, 14 and 15, 16 and 17, and 18 and 19.

Chapter 3.1.2.3 - Conclusions

This is a remarkably interesting board for this project due to its control connection capabilities, the higher analog read precision and the native USB support with ability to rely on the core HID library. On the other hand, the high entry price, its big footprint, and the 3.3 volts limit are drawbacks to be considered. This board could prove harder to work with than other options while its major advantage, its digital pin array, would not be of use unless a complex setup is constructed which may be far from the scope of the prototype. Fortunately, once a software is developed for a board with similar capabilities, the acquired knowledge could be translated to this board to enable for higher end controllers.

Chapter 3.1.3 - Mega 2560 Rev 3

Information from [8] and [9].

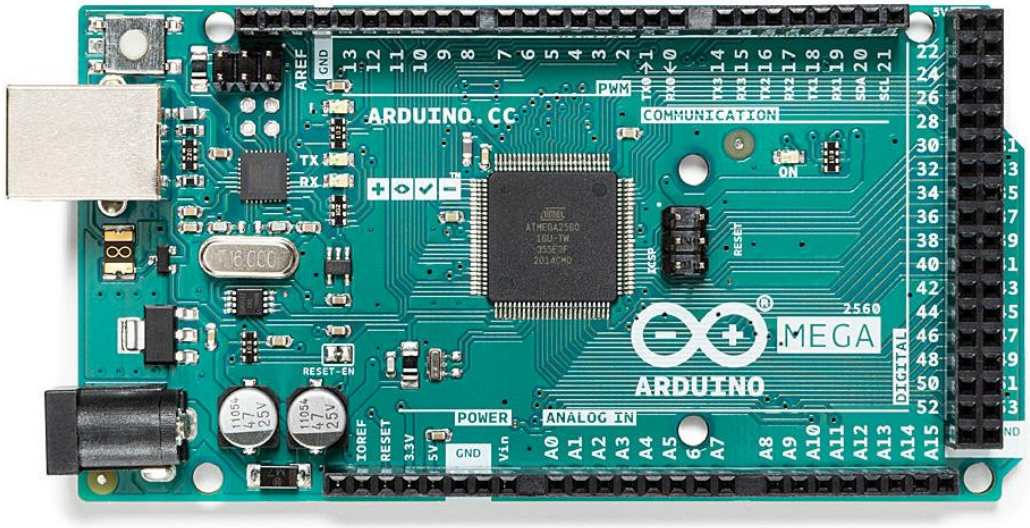

Figure 6 - Arduino Mega 2560 Rev 3

The Mega 2560 is according to Arduino the board with both the greatest digital and analogic I/O. Unfortunately, as we may see in the following overview that is where its advantages conclude and drawbacks from both previous boards arise.

Chapter 3.1.3.1 - Technical Specs

Board	**Name**	Arduino® Mega 2560 Rev3
	SKU	A000067
Microcontroller	ATmega2560	
USB connector	USB-B	
Pins	**Built-in LED Pin**	13
	Digital I/O Pins	54
	Analog input pins	16
	PWM pins	15
Communication	**UART**	Yes, 4
	I2C	Yes
	SPI	Yes
Power	**I/O Voltage**	5V
	Input voltage (nominal)	7-12V

	DC Current per I/O Pin	20 mA
	Supported battery	9V battery
	Power Supply Connector	Barrel Plug
Clock speed	**Main Processor**	ATmega2560 16 MHz
	USB-Serial Processor	ATmega16U2 16 MHz
Memory	**ATmega2560**	8KB SRAM, 256KB FLASH, 4KB EEPROM
Dimensions	**Weight**	37 g
	Width	53.3 mm
	Length	101.5 mm

Table 3 - Arduino Mega 2560 Rev 3 Specs

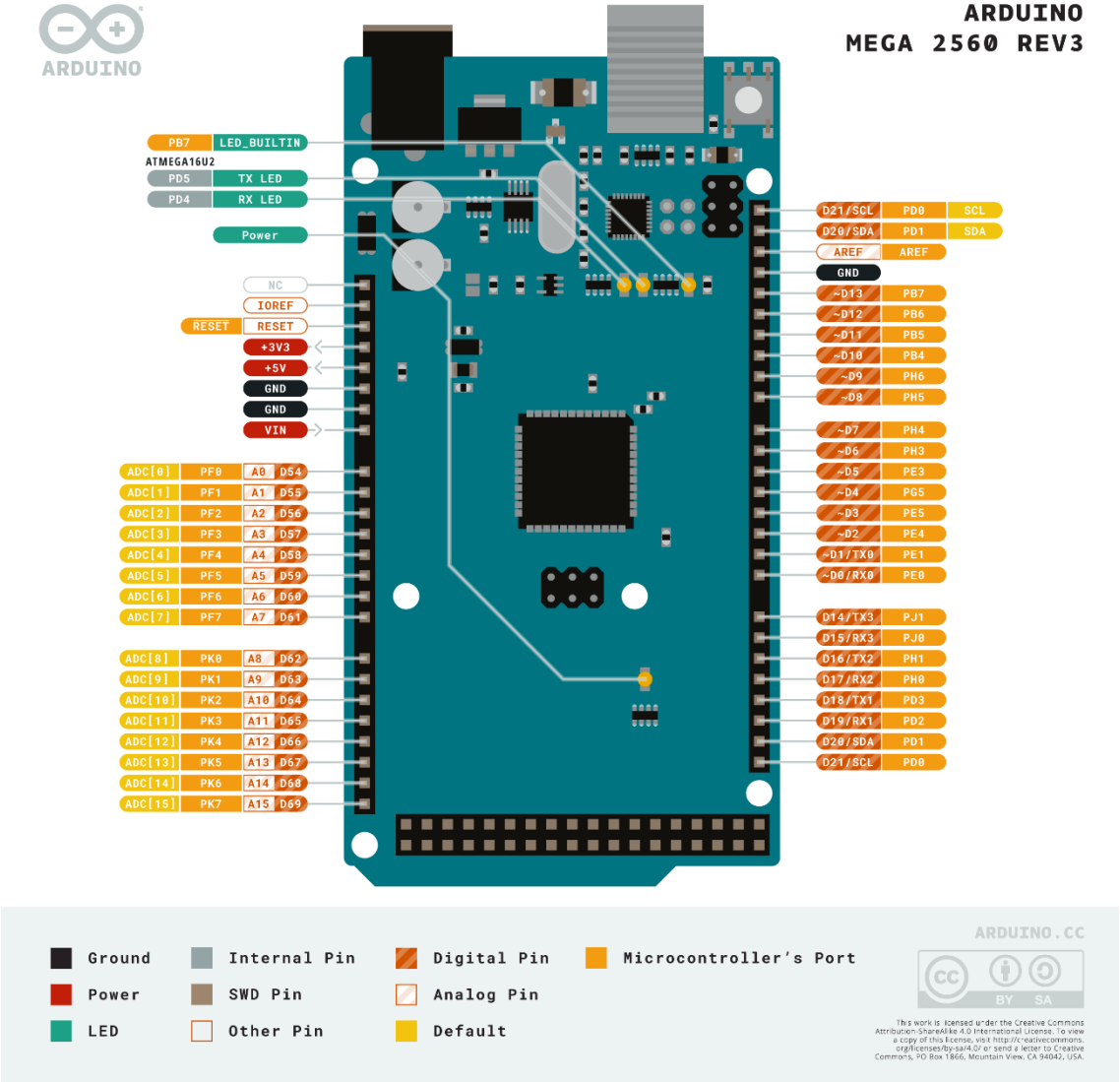

Figure 7 - Arduino Mega 2560 Rev 3 Pinout 1

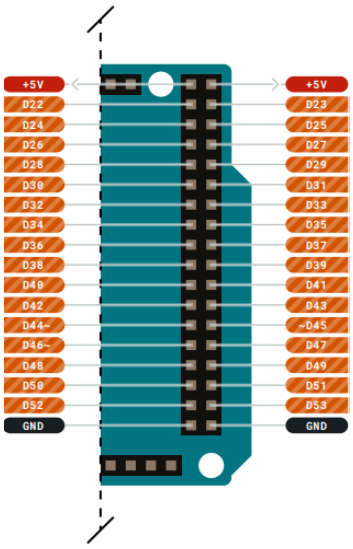

Figure 8 - Arduino Mega 2560 Rev 3 Pinout 2

Watching the specs sheet, pin out and commentary by Arduino on the boards capabilities we can extract that this board while promising at first due to its I/O it can be summarized as a bigger Uno. It has many times the I/O of the Uno and shares the big footprint of the Due, but it does not have the USB connectivity of the Due relying in the same serial over USB through a co-processor as the Uno. Its only real advantage over the Due would be the use of a 5 volts reference that could make the electronics easier to implement.

As stated before, it shares the 53.5 by 101.5 milimeters footprint with the Due, it also shares its price at 35€ but it does not provide the advantages that may prove vital for the project.

Chapter 3.1.3.2 - Theoretical limits for the application

As for the other boards, the limits regarding the existing pins are considered and not the real limits found by testing.

- Up to 16 axes through the analog pins.
  - Reading precision of 10 bits resulting in 1024 possible values.
  - Reading takes 100 microseconds according to Arduino reference, resulting in an approximated max sample rate of 10000 times a second. [5]
- Up to 70 digital controls if every digital and the 12 analog pins were used.
  - As with other boards the use of some pins would disable their use as serial I/O, in pairs, 0 and 1, 14 and 15, 16 and 17, and 18 and 19.

Chapter 3.1.3.3 - Conclusions

This board being an amalgamation of the drawbacks of both previous boards and with its advantage being hindered by then it can be determined to be inadequate with no further comment.

Chapter 3.1.4 - Nano

Information from [10] and [11].

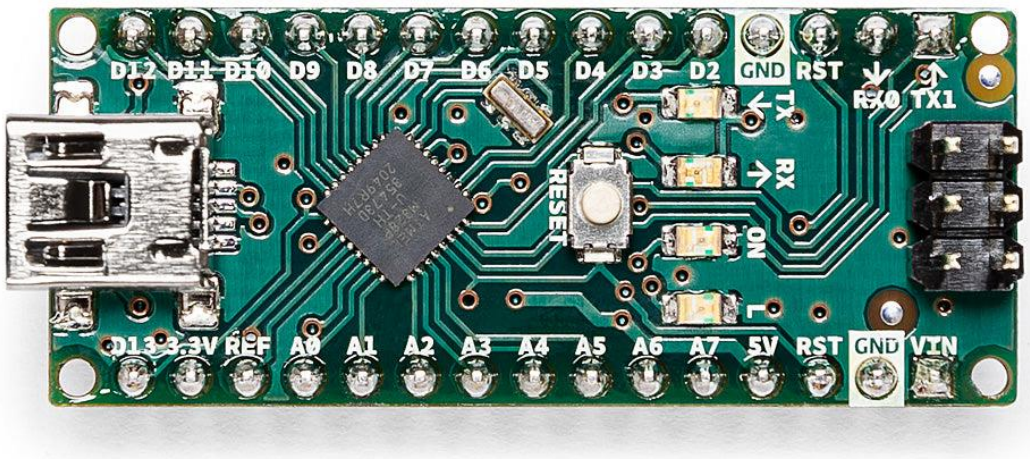

Figure 9 - Arduino Nano

This board was included for consideration as according to the Arduino documentation the Nano and its derivatives constitute a distinct family from the "Classic" boards. Still, after the analysis this board fell in a close position to the Mega 2560 being functionally a side grade to the Uno, this time in compact size.

Chapter 3.1.4.1 - Technical Specs

Board	**Name**	Arduino® Nano

	SKU	A000005
Microcontroller	ATmega328	
USB connector	Mini-B USB	
Pins	**Built-in LED Pin**	13
	Digital I/O Pins	14
	Analog input pins	8
	PWM pins	6
Communication	**UART**	RX/TX
	I2C	A4 (SDA), A5 (SCL)
	SPI	D11 (COPI), D12 (CIPO), D13 (SCK). Use any GPIO for Chip Select (CS).
Power	**I/O Voltage**	5V
	Input voltage (nominal)	7-12V
	DC Current per I/O Pin	20 mA
Clock speed	**Processor**	ATmega328 16 MHz
Memory	**ATmega328P**	2KB SRAM, 32KB flash 1KB EEPROM
Dimensions	**Weight**	5gr
	Width	18 mm
	Length	45 mm

Table 4 – Arduino Nano Specs

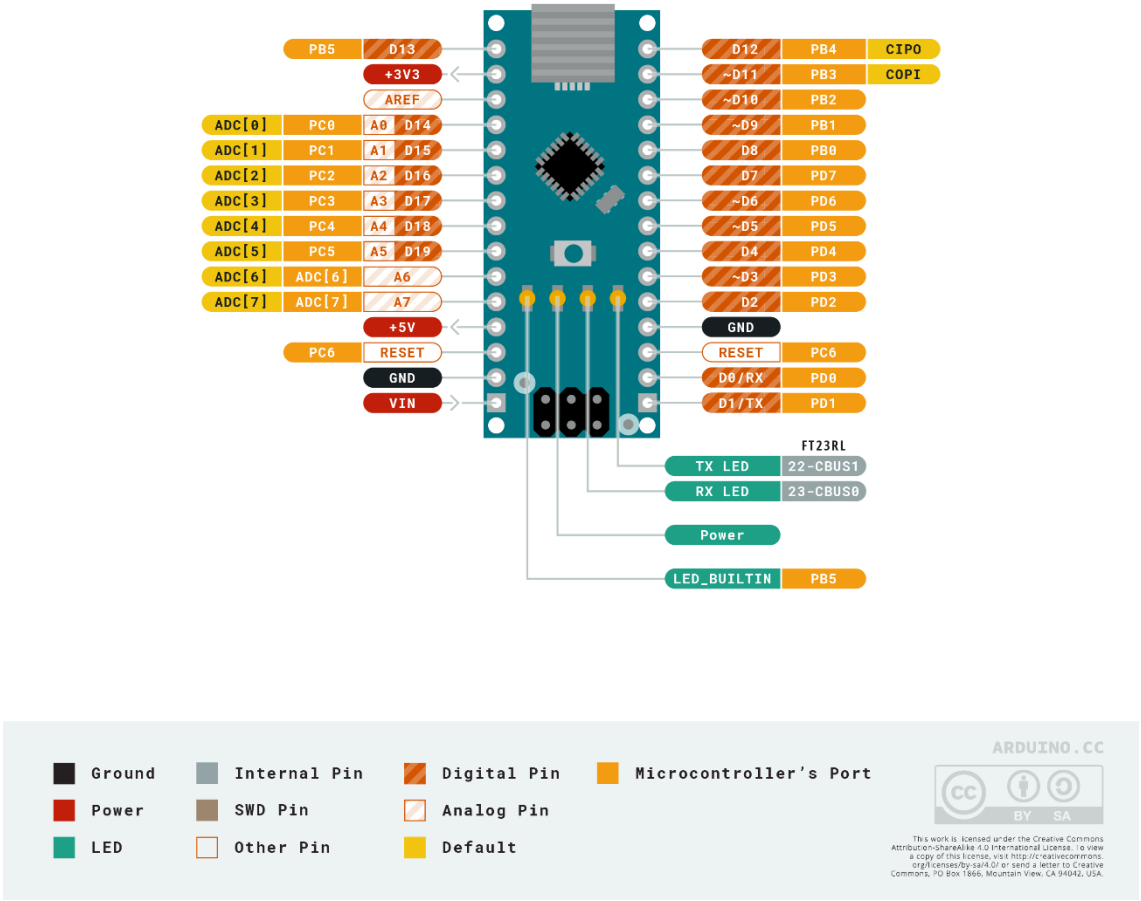

Figure 10 - Arduino Nano Pinout

This board presents the same amount of digital I/O pins as the Uno with the advantage of having 2 additional exclusively analog inputs for a total of 8. Other than that, the relevant specs stay the same as in the Uno but in a smaller package and with the minor difference of using a different co-processor, an FTDI FT232RL.

The board has a footprint of 18 by 45 milimeters and has its connections prepared to be directly embedded into a breadboard instead of using wires. Its retail price is 18€ thus being one of the most affordable boards.

Chapter 3.1.4.2 - Theoretical limits for the application

As for the other boards, the limits regarding the existing pins are considered and not the real limits found by testing.

- Up to 8 axes through the analog pins.
  - Reading precision of 10 bits resulting in 1024 possible values.
  - Reading takes 100 microseconds according to Arduino reference, resulting in an approximated max sample rate of 10000 times a second. [5]
- Up to 20 digital controls if every digital and the 6 digital I/O able analog pins were used.

- Digital pins 0 and 1 could be used for serial I/O limiting the button amount, pin 13 can also be left free for the built in LED to be used.

Chapter 3.1.4.3 - Conclusions

The Nano does not prove to be an adequate board for the same factor the regular Uno is not suitable as well, the lack of native USB communication limits the development for the use of only the serial monitor that may not be enough for the task at mind. Apart from that, its breadboard ready design means that while more compact, it inevitably will take space in a breadboard presenting challenges for the prototype assembly.

Chapter 3.1.5 - Leonardo

Information from [12] and [13].

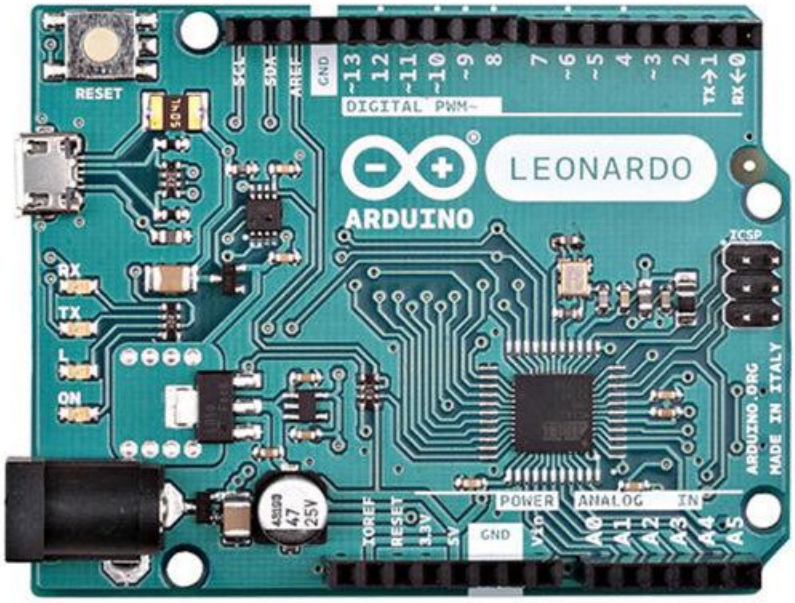

Figure 11 - Arduino Leonardo

This board, unlike the previously commented Due, could really an update over the base Uno, it has the same footprint, digital I/O and analog I/O in the same placement, it would be a carbon copy if not for the selection of an ATmega32u4 main processor with native USB connection that does not need a co-processor for neither the serial communication nor the programming. The native USB connection supports the use of the core HID library the target solution could be developed on.

Chapter 3.1.5.1 - Technical Specs

Board	**Name**	Arduino® Leonardo
	SKU	A000057
Microcontroller	ATmega32u4	
USB connector	Micro USB (USB-B)	

Pins	**Built-in LED Pin**	13
	Digital I/O Pins	20
	Analog input pins	12
	PWM pins	7
Communication	**UART**	Yes
	I2C	Yes
	SPI	Yes
Power	**I/O Voltage**	5V
	Input voltage (nominal)	7-12V
	DC Current per I/O Pin	10 mA
	Power Supply Connector	Barrel Plug
Clock speed	**Processor**	ATmega32U4 16 MHz
Memory	**ATmega32U4**	2.5KB SRAM, 32KB FLASH, 1KB EEPROM
Dimensions	**Weight**	20 g
	Width	53.3 mm
	Length	68.6 mm

Table 5 – Arduino Leonardo Specs

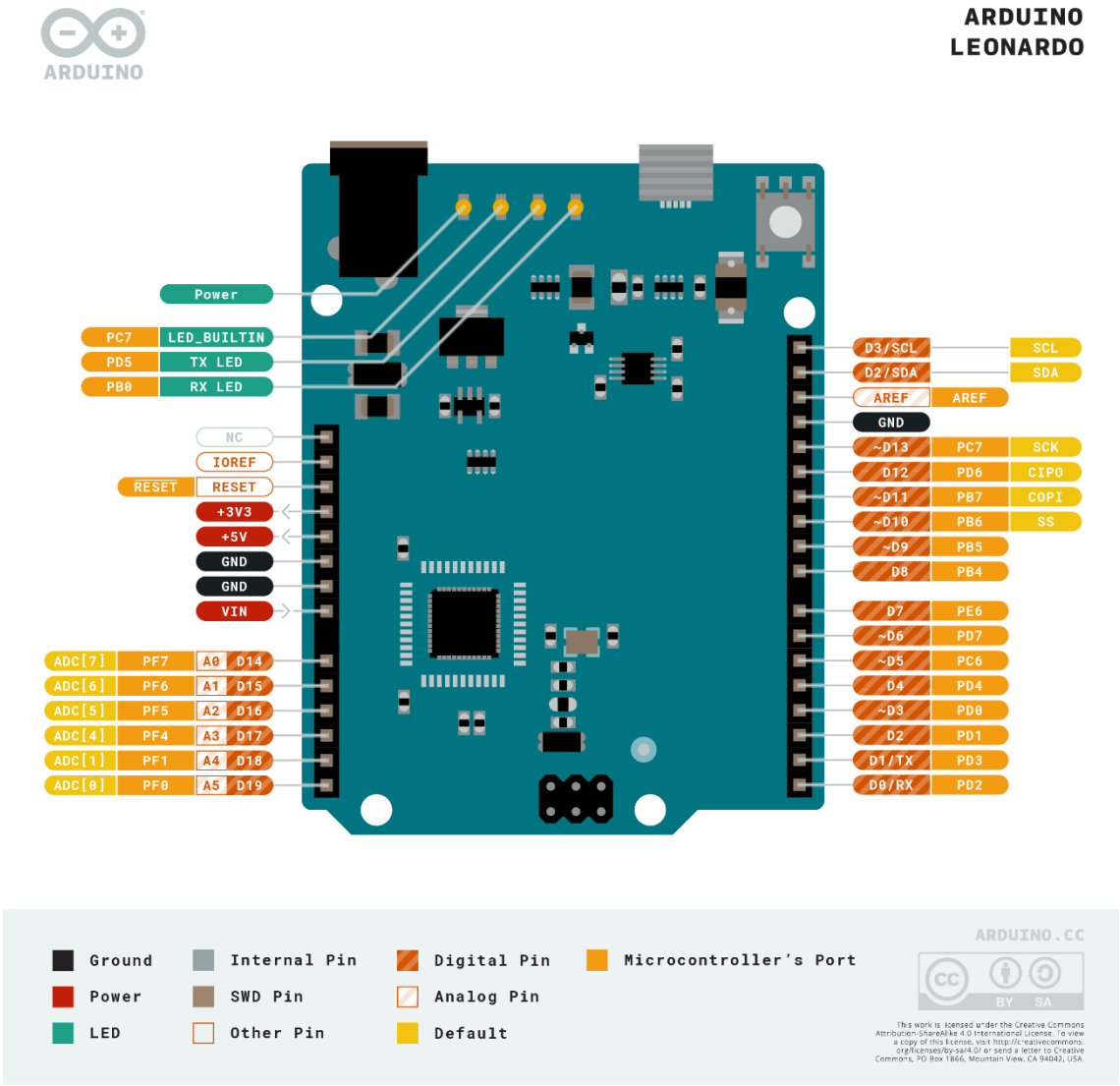

Figure 12 - Arduino Leonardo Pinout

The Leonardo presents a good base to develop on with the footprint of a standard board with headers and the regular pins. But, unlike the Uno it presents a micro-USB connection with the full functionalities of a USB 2.0 connection and the support of the core HID library by its ATmega32U4 which also works at 5 volts for the analog reference.

The board has the regular 53.3 by 68.6 milimeters footprint with the same mounting holes and power supply connector distribution as the Uno with the only difference in that regard being the use of the micro-USB connection. The board has a retail price of 18€ as well being among the most affordable ones.

Chapter 3.1.5.2 - Theoretical limits for the application

As for the other boards, the limits regarding the existing pins are considered and not the real limits found by testing.

- Up to 6 axes through the analog pins.
  - Reading precision of 10 bits, meaning 1024 possible values.
  - Reading takes 100 microseconds according to Arduino reference, resulting in an approximated max sample rate of 10000 times a second. [5]
- Up to 20 digital controls if every digital and the 6 analog pins were used.

- Digital pins 0 and 1 could be used for serial I/O limiting the button amount, pin 13 can also be left free for the built in LED to be used.

Chapter 3.1.5.3 - Conclusions

This is a very adequate board as it has every desirable quality such as a standard footprint and headers for the ease of assembly of a prototype, affordability, and native USB connection with little compromise. The only drawback in comparison with the other viable boards will be a reduced I/O both in the analog and digital side, this on the other means that the software developed for this board should work on the others directly and could be expanded while there is no guarantee that if it was directly developed for a board like the Due it could work in the Leonardo. It would on another stage of the project or as a possible expansion be viable to modify the software to make use of the Due extended I/O.

Chapter 3.1.6 - Micro

Information from [14] and [15].

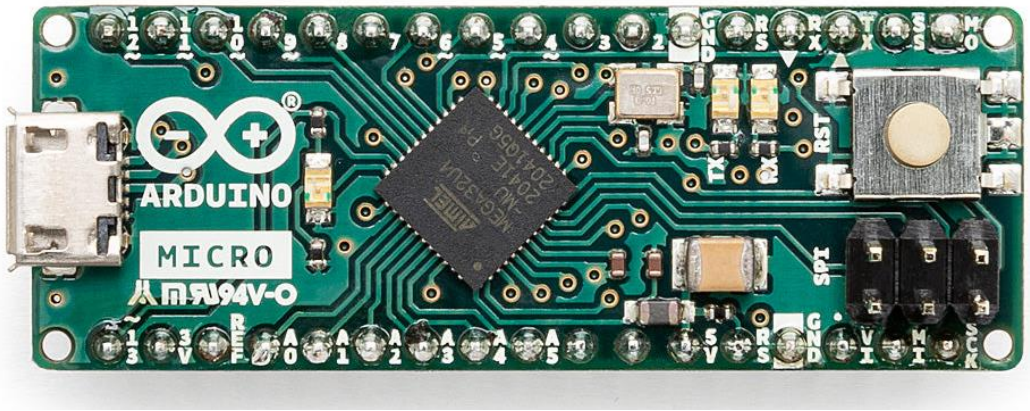

Figure 13 - Arduino Micro

Another ATmega32U4 board just like the previously commented on Leonardo featuring similar capabilities but with a very differing packaging, much more compact and presenting male pins instead of female ones. If used for the project this board would also bring the advantages of a native USB connection able to use the HID library.

Chapter 3.1.6.1 - Technical Specs

Board	**Name**	Arduino® Micro
	SKU	A000053
Microcontroller	ATmega32u4	
USB connector	Micro USB	
Pins	**Built-in LED Pin**	13
	Digital I/O Pins	20
	Analog input pins	12
	PWM pins	7

Communication	**UART**	Yes
	I2C	Yes
	SPI	Yes
Power	**I/O Voltage**	5V
	Input voltage (nominal)	7-12V
	DC Current per I/O Pin	10 mA
Clock speed	**Processor**	ATmega32U4 16 MHz
Memory	**ATmega328P**	2.5KB SRAM, 32KB FLASH, 1KB EEPROM
Dimensions	**Weight**	13 g
	Width	18 mm
	Length	48 mm

Table 6 - Arduino Micro Specs

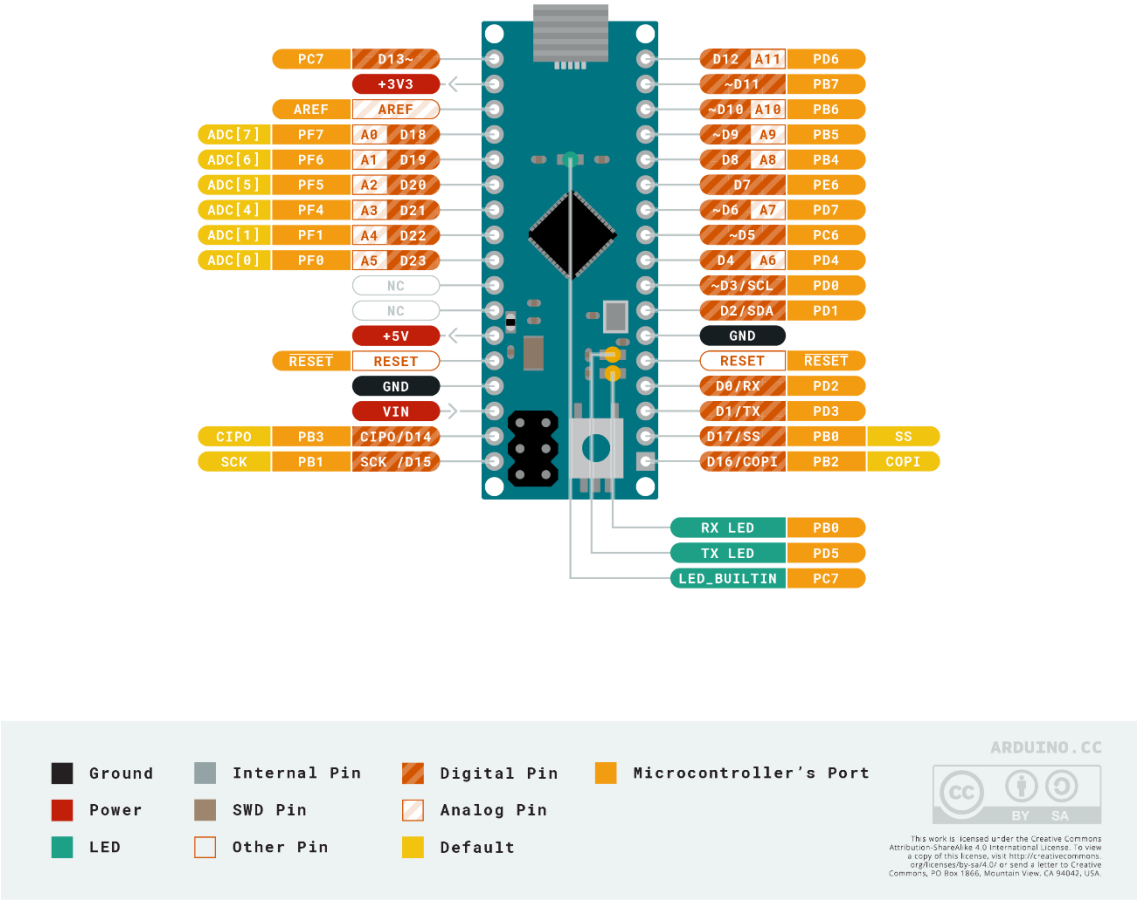

Figure 14 - Arduino Micro Pinout

The Micro presents specifications close to those of the Leonardo sharing the same processor and almost the same digital and analogic I/O pins with the little change that some more of the pins support the analog read function for a total of 12 out of 20 instead of 6 out of 20. Other than the only relevant differences might be the smaller footprint at only 18 by 48 milimeters that would make it a very interesting board for compact assemblies such as handheld controllers, and the lack of female pin connectors in any version of the Micro that could result in added complications when assembling a prototype. At the time of writing these lines this board can be found for 18.9€ in the official Arduino store.

Chapter 3.1.6.2 - Theoretical limits for the application

As for the other boards, the limits regarding the existing pins are considered and not the real limits found by testing.

- Up to 12 axes through the analog pins.
  - Reading precision of 10 bits, meaning 1024 possible values.
  - Reading takes 100 microseconds according to Arduino reference, resulting in an approximated max sample rate of 10000 times a second. [5]
- Up to 20 digital controls if every digital and the 12 analog pins were used.

Digital pins 0 and 1 could be used for serial I/O limiting the button amount, pin 13 can also be left free for the built in LED to be used.

Chapter 3.1.6.3 - Conclusions

This could be a solid option if an objective of the development was to make a refined prototype that could be presented as a final fully integrated product, but in the proposed scope the lack of female pin connectors and the small footprint would result in added challenge. It offers characteristics that make really adequate otherwise.

Chapter 3.2 - Selected board, compatibilities, and prototype hardware

After analysing the previously mentioned board the final conclusion is that the most adequate board for the scope of this project is the Leonardo due to it presenting the native USB connection and being basic board in its connectivity meaning that software developed for this board would not only work in boards with more extended connectivity, but it would also be easy to modify to use that extended connectivity. Software developed for Leonardo should also work in Due and Micro, but not necessarily the other way around in pins present on those boards but not on the Leonardo are used.

The board was finally sourced from the Spanish Amazon store and sold directly by Arduino and handled by Amazon at a price of 21.78€ with free shipping.

Chapter 3.2.1 - Additional prototype hardware

According to the capabilities of this board, it should be possible to manage 6 analogic axis and 14 digital inputs, but it would be adequate to leave some pins free for extended functionalities such as an output LED for debugging or status indication and an additional one for a switch or jumper that allows for precision selection. It would be possible to use a different balance between buttons and axis, but 6 axes are adequate as the correspond with a configuration of 2 joysticks, and X and Y axis for each, and 2 pedals or analog finger triggers being a configuration valid for most modern handheld controllers or a versatile custom setup.

The prototype hardware capabilities are expected to be just enough to assess the software function, and as such no special requirements other than being adequate for quick prototyping and presenting the most basic and regular function. If the software is correctly developed the use of differing components should not be an issue.

With this decision hardware covering those inputs was searched for in many online distributors. At the end, the following packages were found in the Spanish Amazon store and are distributed directly by them having free shipping and enforcing some post sell requirements for sellers.

Chapter 3.2.1.1 - ELEGOO Upgraded Electronic Components E3 kit compatible with Arduino IDE

As listed in the store: ELEGOO Kit Mejorado de Componentes Electrónicos con Módulo de Alimentación, Placa de Prototipos (Protoboard) de 830 Pines, Cables Puente, Potenciómetro, STM32, Raspberry Pi, Compatible con Arduino IDE

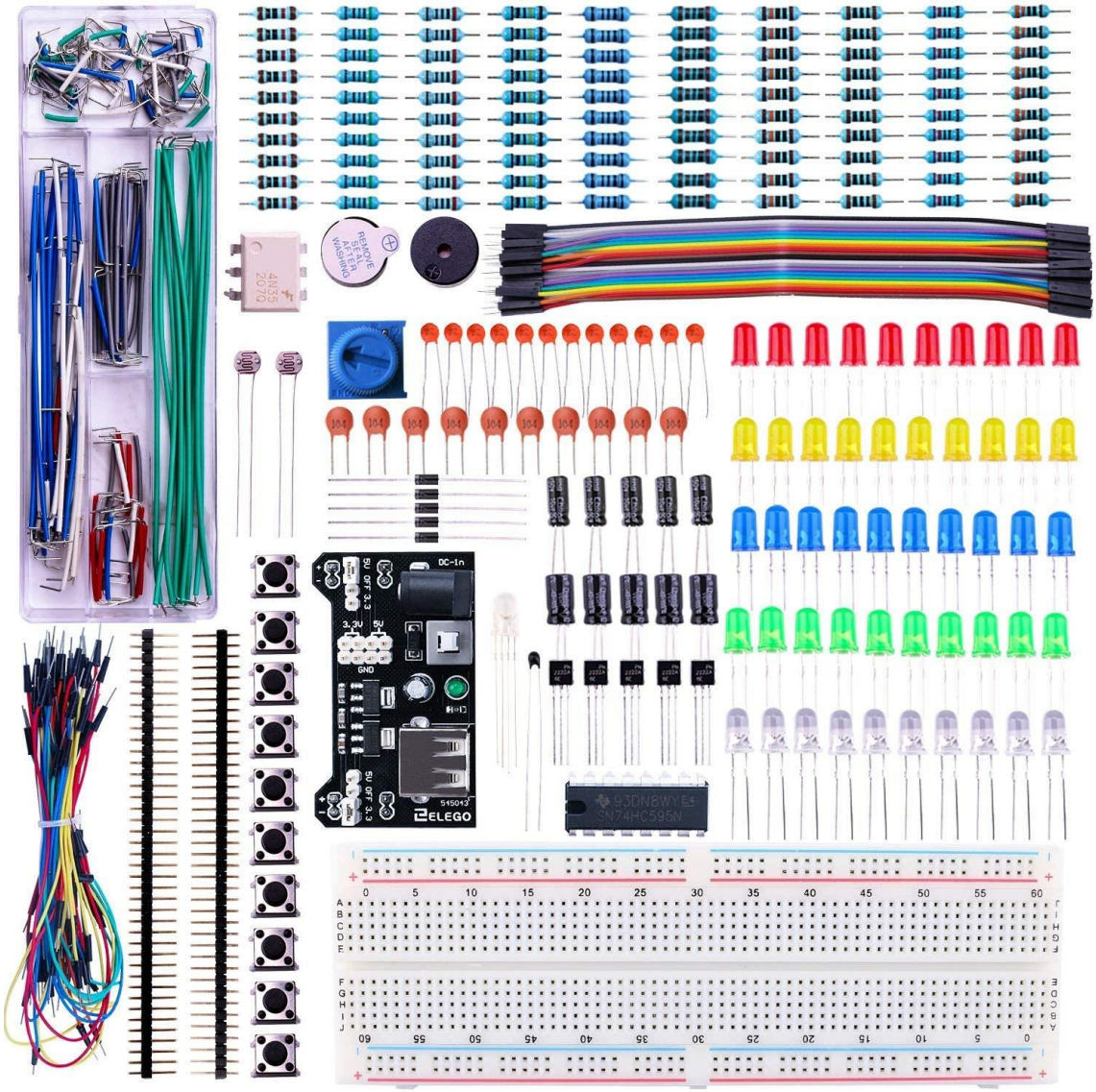

Figure 15 - Chapter 3.2.1.1 - ELEGOO Upgraded Electronic Components E3 kit

This kit was selected as it included many of the parts that would later be used in the prototype assembly. While others are included, those to be used for the prototype are the following:

- Breadboard, serving as assembly base.
- 10 buttons, for 10 digital inputs.
- Rigid jumper wires, adequate for low profile connections.
- Flexible jumper wires, used for long connections.
- Female-Male Dupont wires, for connecting the joysticks.

The kit covered many of the needs for the prototype and provided some spares for the wires that can get easily damages, in the case any of the required more limited parts such as the buttons arrived damaged it would be possible to obtain a refund or replacement during the first 30 days covering the most probable failure cause for this kind of hardware. The entire kit was acquired for 14.99€ being more economic than sourcing the pieces separately, which would only be practical if a spare was needed at a later date.

Chapter 3.2.1.2 - WINGONEER PS2 styled Joystick, pack of 5

As listed in the store: WINGONEER 5PCS PS2 Joystick Juego de control XY de dos ejes Módulo de desbloqueo de joystick para Arduino

Figure 16 - WINGONEER PS2 Styled joystick

These joysticks were selected for two purposes, firstly, each of them includes two potentiometers with separate output pins for one 5 volt and one ground pin, meaning that each can handle 2 axes. They also present a button like those included in the previously commented components kit, this button has its own pin and shares the ground with the potentiometers. Pins are located in such a way that they can be connected to the breadboard using the Female-Male Dupont wires in packs of 5 resulting in a quite rigid band. For the prototype purposes 3 of them would be used wiring both axes in each for a total of 6 and then wiring the button for 2 of them to complement the 10 buttons for a total of 12 digital inputs. Like the electronic components kit, they can also be replaced easily during the first 30 days in case of failure and being a pack of 5 means there will be two spares. The cost of the pack was 9.99€

Chapter 4 - Activities planning and management

In this chapter we specify the activities carried for the project completion. Due to the nature of the project, mostly investigative and to be carried by an individual with the objective of a single main product, a prototype loaded with the developed software, there is no personnel management involved in the planification and the times may not be estimated for the reduced knowledge in the matter prior to the development. The carried tasks are the following:

- Investigating commonly available Arduino boards and their capabilities as part of a viability study.
- Gathering information about the HID standard.
- Decide on a board.
- Determine controls to be implemented on the board.
- Study on Arduino software development.
- Software design.
- Prototype design.
- Acquisition the board and electronics for the prototype assembly.
- Parallel assembly of the prototype and incremental software development.
- Prototype testing and acceptance.
- Study and documentation of possible software distribution.

As part of the planning a reduced risk management was performed exclusively for matters related to the development and the hardware involved in the prototype, no personnel management is involved. The risk identified was the possibility of getting faulty hardware or not receiving it in a reasonable time, either if it was a long delivery time or a replacement was to be ordered both would impact the time taken for the development as the material for the prototype would be missing and the necessity of having a protype, even if partial, for testing matters. For this reason, it was decided to prioritize getting the materials from a trusted vendor than locating a vendor adequate for bulk orders intended for mass production. After all, the prototype is intended to be a one off specifically made for this project and not as product to be sold later. The final solution to the issue was to buy every product through the Prime service of the Spanish Amazon store that warrants that the materials are delivered quickly, and it would be easy to obtain replacement in case of failure.

Chapter 5 - Solution software design

In this section we comment the design of the software solution, both defining what it is supposed to do to fulfil its purpose and then explaining the carried design to accomplish it.

Chapter 5.1 - HID, what it is and what the board does

The objective of the developed software is for the board to behave as a Human Interface Device, meaning it communicates data through the USB as a indicated by the Device Class Definition for HID which extends the USB Specification, its objective is to allow manufacturers to create devices compatible with USB and indicates how a driver should interpret the information communicated by the device. For the scope of this project, we are mainly interested in the former.

For a device to work as a HID over USB it is required for it to be able to first describe itself to the host system, the computer the board is connected to in our case, what information it will report and in which way, communicate its descriptor. Once this is done the communication of the relevant data will begin to be sent as the device is pooled, this information is composed according to the descriptor in one or several different reports. Fortunately for the development of the project, most of the USB communication can be managed by the core HID library for Arduino, which relying in the PlugableUSB library, allows a compatible board, such as the selected Leonardo and some others, to communicate with a host device as a HID compliant device by managing the correct sending of the descriptor and subsequent reports when pulled for them by the host.

Chapter 5.1.1 - The HID library for Arduino and its use

This is a core library for the Arduino boards of model Due, Leonardo, Micro and Zero. It works over the PlugableUSB library and manages most of the tasks a HID compliant device has to perform. This library has its code published in the ArduinoCore-avr GitHub repository [16] that has been consulted for better understanding of the inner workings of the library as the official guides on the Arduino reference page do not show the entire picture in how the library must be used. A brief explanation in the use of the library for the creation of "PluggableHID" can be found in the wiki for the Arduino repository [17] also on GitHub.

The library exposes two public methods apart from its constructors. By using these two methods with the proper input data it is possible to create the software for a HID compliant device. The development would then revolve around correctly feeding them.

- *void AppendDescriptor(HIDSubDescriptor* node)*

  This method is not referenced in the Arduino reference page. During the development, its usage was deduced by analysing the source code of the library. At the time of writing these lines a concrete guide by Arduino was found in a different repository, but only including how to use this method and not the second one that is later commented in the chapter *The HID-Controller library implementation*.

  The purpose of this method is to set up a descriptor to be used when carrying the USB communication. For this matter it is necessary to first build the descriptor and provide it to this method as parameter. The topic of building the constructor is explained in a later section of this document.

- *int SendReport(uint8_t id, const void* data, int len)*

The objective of this method is to trigger the communication of the report indicated in the first parameter, id. It also requires the data of the report itself, data, and its length, len. The data parameter has to receive a pointer to where the report you want to communicate is, len

asks for its length in bytes. The method then relies this data to endpoints of PlugableUSB for it to be communicated to the host.

The major difficulty of using this method comes from the way the report has to be built and matching each report with its id correctly when passing the data, as there is no method for sending every report at the same time, each has to be created independently. More specifics on how to handle a report can be found in the section.

The use of this library signifies a great starting point for the development as it provides a native way for the boards to carry most of the required functionality expected from then in the scope of the project. Using a native library generally warrants the issue free and quick working that may be required for the application while reducing development times.

Chapter 5.1.2 - The creation of the descriptor

In this section we comment the definition of a HID descriptor that our board will use to define the host computer its capabilities. First, we have to decide what inputs we will finally include and then construct an adequate constructor according to the HID requirements. The inputs we dedicate for the controller part are 6 analog axes and 12 digital inputs, out of those we decide to dedicate 4 to implement a hat switch, also known as a DPAD or POV control, leaving us with 8 buttons.

The report has to be composed by the conjunction of the hexadecimal words according to guides published in the USB Implementers Forum website, more specifically the section dedicated to HID developers. The guides give far more detailed than required for the scope of the project explanations about the topic at hand and they include the much needed reference tables matching every usage you want to include in your report to its corresponding hexadecimal words. The descriptor was composed using the Device Class Definition for Human Interface Devices [18], mostly the section 6 explaining descriptors and the Appendix E examples, and the HID Usage Tables for Universal Serial BUS (USB) [19]. For better understanding some additional examples of descriptors extracted from relevant devices such as modern controllers were consulted, they were compiled in a GitHub repository [20].

Firstly, it is necessary to provide some information about the report itself, so we indicate that we are going to make a report for a Joystick, we have to first specify the usage page (0x05) it belongs to, Generic Desktop Page (0x01), we indicate the usage (0x09) inside the page, joystick (0x04) and then start a collection (0xa1) that groups all the features we want to report for our joystick or controller, in a modern system they both are seen as “Gaming Devices” and its purposes basically indiscernible, both will be handled in a similar way, but for this specific use case where several axes are reported with accuracy the definition of Joystick seems more adequate for the HID descriptor according to the reference. This collection contains all the features of the joystick, so it is qualified Application (0x01), the reference states to use at least a report for each application collection, which is also needed for the HID library to communicate our report. Being our only report we use the report ID 1, this is not compulsory but preferable in case we would add further reports in the future. Extracted from the final code the result for this part is the following:

```
static const uint8_t hidDescriptor[] PROGMEM = {
    //Controller with 8 buttons, 4 16(10)bit axis, 2 8bit axis and a 2
DPADs (1 used)
    0x05, 0x01,                         // Usage Page (Generic Desktop)
    0x09, 0x04,                         // Usage (Joystick)
    0xa1, 0x01,                         // Collection (Application)
    0x85, 0x01,                         //   Report ID
```

With the report started we now have to include our controls in the report, we start with the 8 buttons that serve as an example for how the inputs have to be define as we have to specify more than use pages and usages, buttons page (0x09) and usages from (0x01) to (0x08) which are the button numbers. Now we are also required to indicate how the data communicated is, we have to indicate the limits for the data reported, 0 to 1 being logical controls, and the number of controls to be reported, 8. Then for every input we want to include we have to declare it as such by including a input line (0x81) afterwards and using the correct word to for the characteristics of our input, for all of them in our report it will be (0x02). This word for input indicates the following:

- Bit 0 → 0 → Data: In contrast to the alternative Constant, this modifier implies the value can be modified instead of being a static read only field.
- Bit 1 → 1 → Variable: Dictates that the value of each input is reported as a single variable with the number of bits indicated in report size instead of the alternative of Array that may group many inputs in an array.
- Bit 2 → 0 →Absolute: Used instead of relative it makes the value reported to be the current absolute value instead of reporting the difference from the previous time it was reported
- Bit 3 → 0 → No Wrap: When No Wrap is used the value reported when reaching the maximum or minimum will just stay there instead of going from one to the other as when you are turning a 360 degrees dial.
- Bit 4 → 0 → Linear: If set to linear this indicates the reported value has not been processed in any way such as non-linearity of dead-zones for joysticks.
- Bit 5 → 0 → Preferred State: When this property is indicated it translates to the control having a default state it will return to when not interacted with, for example when a joystick returns to its centered position when not pushed in any direction.
- Bit 6 → 0 → No Null position: The control is always sending data.
- Bit 7 → 0 → Non-volatile: This property tells the host system that the value may not be modified by the host, only by the control.

```
// 8 Buttons
0x05, 0x09,                    //   Usage Page (Button)
0x19, 0x01,                    //   Usage Minimum (Button 1)
0x29, 0x08,                    //   Usage Maximum (Button 8)
0x15, 0x00,                    //   Logical Minimum (0)
0x25, 0x01,                    //   Logical Maximum (1)
0x75, 0x01,                    //   Report size (1)
0x95, 0x08,                    //   Report Count (8)
0x81, 0x02,                    //   Input (Data,Var,Abs)
```

Next, we define our axes, this case came to be a bit particular. To begin, in the way the usages have to be declared, each axis needs its usage to be declared individually unlike with the buttons where the minimum and maximum values for a range are indicated. To continue, when data is obtained from different physical locations in the device, the documentation recommends grouping then in many Collections of the Physical kind, for example grouping all the axes reported by a joystick in one collection. In our case the final result will just expose methods to give the values to each axis, button or the DPAD while not knowing how the complete setup is constructed, it is not relevant for the inner workings of the software. We could go with the grouping relevant for the prototype, but that is only the case of this instance, with all this in mind we decide adequate to group all the axes in a single collection.

Finally, during the development phases of the software and in conjunction with the test prototype, a limitation was found in the report that affected the report and the rest of the software

in relation with the axes. At first it was attempted to report all the 6 axes with a 16 bit resolution to make use of the maximum precision of 10 bits the board provides. Unfortunately, a hard limit was found where the board would only report 5 of the axes and did not communicate any data for the sixth one. It was then decided that both Z axes would be reported with a resolution of just 8 bits solving the issue, as such, they have to be declared apart as bellow.

```
    // 4 16bit(10) Axis
    0x05, 0x01,                         //   Usage Page (Generic Desktop)
    0xa1, 0x00,                         //   Collection (Physical)
    0x09, 0x30,                         //     Usage (X)
    0x09, 0x31,                         //     Usage (Y)
    0x09, 0x33,                         //     Usage (Rx)
    0x09, 0x34,                         //     Usage (Ry)
    0x15, 0x00,                         //     Logical Minimum (0)
    0x27, 0xFF, 0xFF, 0x00, 0x00,       //     Logical Maximum (65534)
    0x75, 0x10,                         //     Report size (16)
    0x95, 0x04,                         //     Report Count (4)
    0x81, 0x02,                         //     Input (Data,Var,Abs)
    // 2 8bit Axis
    0x09, 0x32,                         //     Usage (Z)
    0x09, 0x35,                         //     Usage (Rz)
    0x15, 0x00,                         //     Logical Minimum (0)
    0x26, 0xFF, 0x00,                   //     Logical Maximum (255)
    0x75, 0x08,                         //     Report size (8)
    0x95, 0x02,                         //     Report Count (2)
    0x81, 0x02,                         //     Input (Data,Var,Abs)
    0xc0,                               //   End Collection
```

To finalise, we add the D-PAD or hat switch to the report, for this case, another peculiarity was found in that it was necessary to include a second D-PAD in the descriptor for the reporting to work, with the D-PAD having 8 possible directions it can be reported with only 4 bits, so a second D-PAD is declared to match a byte in total size and allow the reporting to operate. This second D-PAD could still be used by modifying the code renouncing to 4 buttons or axes in our board or being added in a Due version of the software.

```
    // 2 DPADs (1 used)
    0x05, 0x01,                         //   Usage Page (Generic Desktop)
    0x09, 0x39,                         //   Usage (Hat switch)
    0x09, 0x39,                         //   Usage (Hat switch)
    0x15, 0x01,                         //   Logical Minimum (1)
    0x25, 0x08,                         //   Logical Maximum (8)
    0x75, 0x04,                         //   Report size (4)
    0x95, 0x02,                         //   Report Count (2). While only
1 report is used, this is necessary to match the bits position
    0x81, 0x02,                         //   Input (Data,Var,Abs)
```

We end the descriptor by closing the Application Collection.

```
    0xc0                                // End Collection
};
```

Chapter 5.2 - Architecture of the solution and inner design

While being based in C and supporting objects, the development of software for Arduino is structurally much simpler than that of regular software. Arduino solutions generally consists in a mostly functional close to a regular Main method conjunction of the setup and the loop methods that are required in *.ino* file. If further functionality is required in can be added by defining functions in the *.ino* file itself or by importing libraries, let it be a core library or a third party one. Libraries can be created by creating at minimum two files, a *.h* file or header file where the definitions of data, functions of classes can be made, and a *.c* or *.cpp* file where the functionality itself is coded in C or C++, respectively. In summary, a complete solution will consist in the .ino file working as Main and then as many header and code file conjunctions as additional libraries are used for the *.ino* to fulfil its purpose.

For our situation we identify that the software side of our board has to perform two tasks. First, interpret the values obtained from the physical controls and secondly communicate them over the USB connection. These two tasks will be performed indefinitely since the moment the device is connected until it is disconnected. Performing the actions will then be carried as part of our loop in *.ino* file. Still, the functionality of communicating the values of our controls over the USB is identified as something that can be extracted to its own library, our adaptation of the HID library for our specific purpose and potentially reused in other solutions. With all the previous statements in mind we can define the design of out solution. In this design some additional data such as the files in which each entity can be found is included.

Due to the way libraries for Arduino have to be coded the entity *ControllerImpl* is added for better understanding of the design, but in the code itself there are no references to that name, this entity consists in the many methods declared as *"Controller.methodName(parameters)"* in the *HID_Controller.cpp* file. It may be considered as an Adapter class from the Object Adapter design pattern while Controller is the target and the HID Library the Adaptee [21].

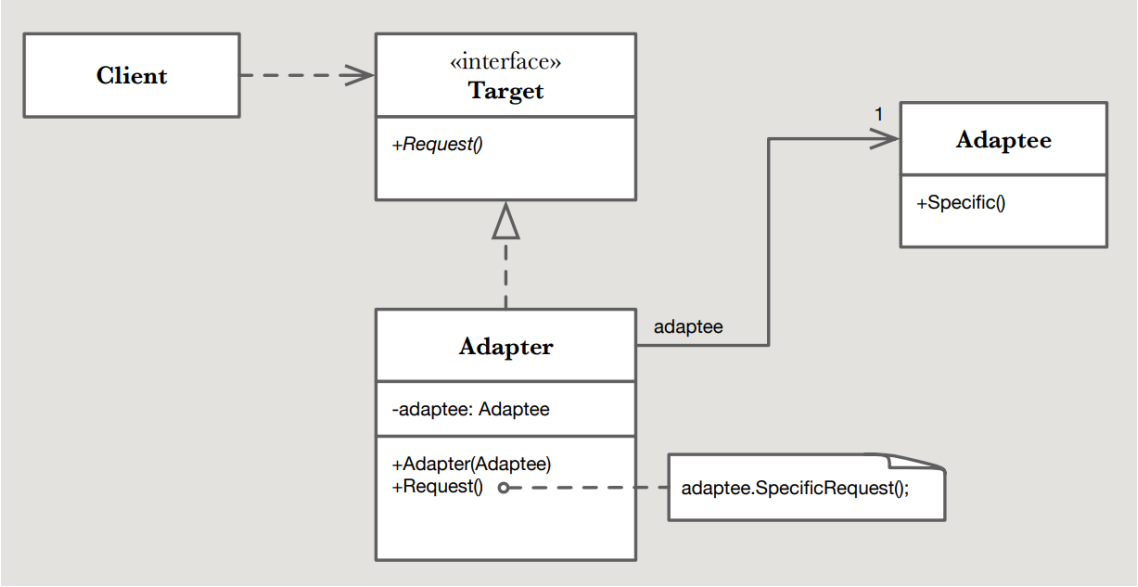

Figure 17 - Adapter design pattern as explained in the Software Design subject

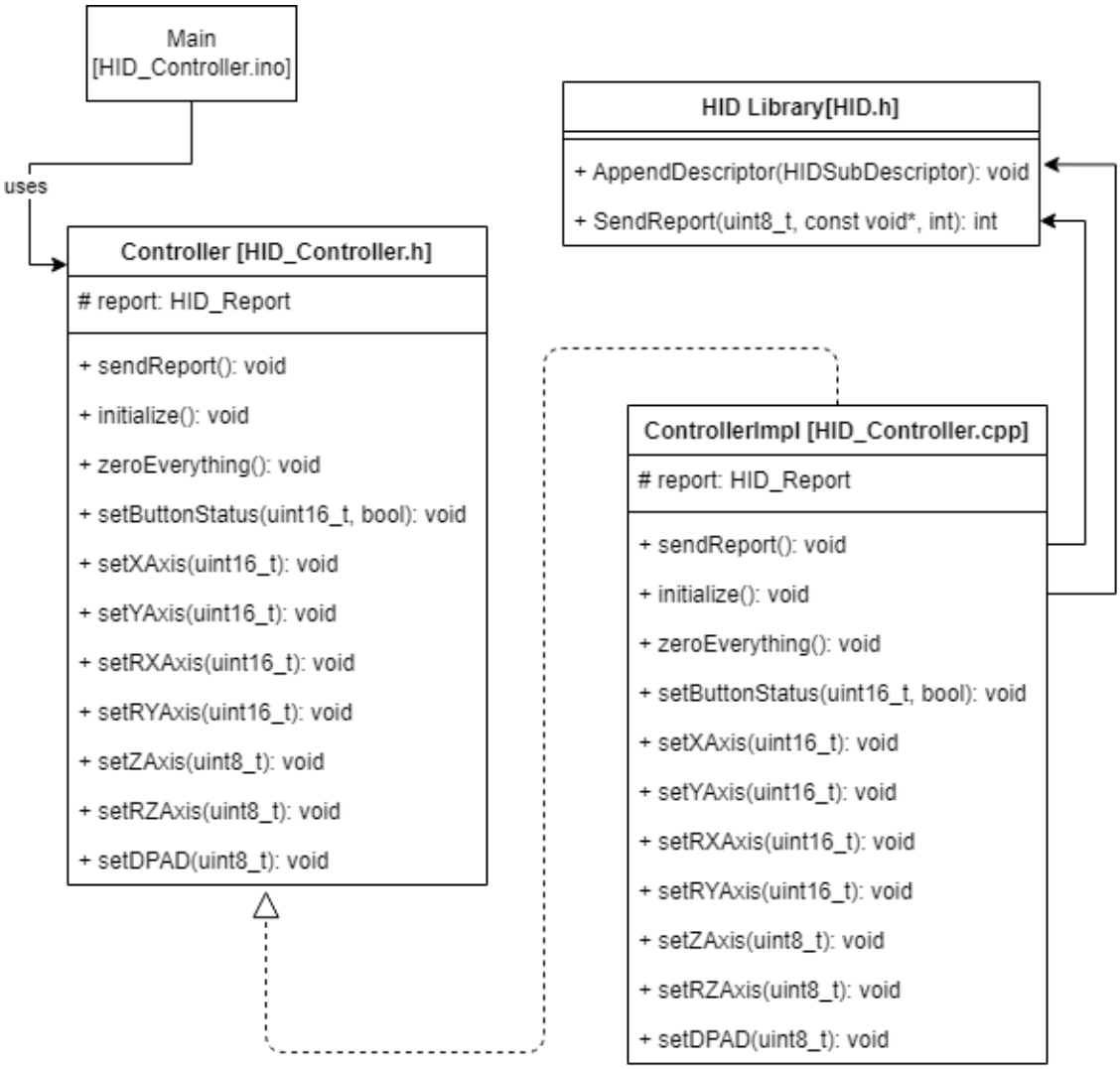

Figure 18 - Software solution design

Chapter 5.2.1 - Setup and loop implementation

The tasks carried in the *.ino* file are already explained in the previous section, now we will deepen in the way they were implemented and how everything is performed.

We start with a block with declaration of variables that will later be used and the inclusion of our *HID_Controller* library explained in the following section.

```
#include "HID_Controller.h"

bool BIT_JUMPER;
Controller controller;
```

Apart from including the library we are declaring the bool variable *BIT_JUMPER* as it will be used to store the value of the pin 12 that will be read once per loop iteration and then used several times when reading the analog axes values.

We now review the setup function. This function will be called once automatically when the device is connected, so we use it to initialize every variable and the pins we will later use.

```
void setup()
{
  pinMode(12, INPUT_PULLUP); // Axis precision jumper
  pinMode(13, OUTPUT);       // Led output for debugging
  pinMode(0, INPUT_PULLUP);  // DPAD-Right
  pinMode(1, INPUT_PULLUP);  // DAPD-Up
  pinMode(2, INPUT_PULLUP);  // DPAD-Down
  pinMode(3, INPUT_PULLUP);  // DPAD-Left
  pinMode(4, INPUT_PULLUP);  // Buttons from 1 to 8
  pinMode(5, INPUT_PULLUP);
  pinMode(6, INPUT_PULLUP);
  pinMode(7, INPUT_PULLUP);
  pinMode(8, INPUT_PULLUP);
  pinMode(9, INPUT_PULLUP);
  pinMode(10, INPUT_PULLUP);
  pinMode(11, INPUT_PULLUP);
  // Send controllers initial report.
  controller.initialize();
}
```

We first initialize the pin 12 as a pullup input to use for our bit precision jumper wire related to the *BIT_JUMPER* variable. Then the pin 13 is initialized as an output so the LED connected to it directly on the board can be used as an output, this was useful during development stages as a debugging output in substitution to the serial monitor that stops working if any of the pins 0 and 1 are used. We initialized the pins from 0 to 11 as pullup inputs, allowing us to wire buttons without the need of an additional resistor, we are using the internal one in the board instead. Using the inputs this way allows for a simpler wiring of the buttons using less components and just requires considering the digital input values will be inverted, high state when not pressed and low otherwise. Finally, we call for the initialize method of our controller library, sending an initial report with every value set to zero.

Before our loop we define two functions that will be used many times on each iteration and thus extracting them allows to prevent code repetition and makes the code itself easier to read. These functions are meant to capture the values of the axes from the analog input of the board and translate them into adequate values for the HID report.

```
int getJoystickValue(int pin)
{
  int pinRead = analogRead(pin);

  if (!BIT_JUMPER)
  {
    pinRead = map(pinRead, 0, 1023, 0, 255);
    return map(pinRead, 0, 255, 0, 65534);
  }
  return map(pinRead, 0, 1023, 0, 65534);
}
int get8BitAxis(int pin)
{
  return map(analogRead(pin), 0, 1023, 0, 255);
}
```

Even though both functions are meant to read an analog axis it was considered necessary to create two different ones depending on the axis precision. In hid devices it is possible to use either 16 or 8 bit precision axis while our board can only operate at 10 bit precision. This presents us a problem to solve in that none of our ranges matches the other. We start with the more complex *getJoystickValue(int pin)*, as understanding this function would make the other trivial in comparison. Firstly, we take the analog pin number we want to read as a parameter so the function can be used for every axis. We then read and store the value of the axis in the native 10 bit resolution, a value from 0 to 1023. Now we are presented with two paths to act depending on the previously commented *BIT_JUMPER* variable. When working with the most basic potentiometers for the axis such as the carbon-carbon ones found in our joysticks, or using low gauge unshielded wires, it is quite common that the read values present a high noise level meaning that even when the control is not being moved the value itself will keep changing between close values. This would translate into shaky controls being sent to the host machine affecting handling in applications such as simulators. For this reason, it was decided to add an 8 bit precision mode even for the 16 bit axes, in 8 bit precision the step from one value to the next is increased reducing the probability of the final values jumping from one to the next, it allows compromising accuracy in order to get a higher precision if the control device used does not provide enough. The value from the pin 12 is read and if found to be at low, the jumper in our case is connected, then the 10 bit value obtained from the analog input is mapped into an 8 bit value before mapping it to a 16 one for the report. Otherwise, it is mapped straight from 10 to 16 bit. By mapping we are referring to the use of the native function map which translates values from a range in one magnitude to a different one linearly, in our case we would be translating into the same percentage of action in the axis but at different precisions. The *get8BitAxis(int pin)* function performs the same action of translating the value of an analog pin into one that can be used for the report but only from the native 10 bit to 8 bit without option.

We finally proceed to comment on the *loop* function. The objective for iteration will be gathering the current values for each control and by the use of our controller library send a report corresponding to them. To start, we capture the values of our axes and the bit precision jumper state for the axes' translation.

```
void loop()
{
  BIT_JUMPER = digitalRead(12);
  // Joysticks
  // Joystick1
  controller.setXAxis(getJoystickValue(0));
  controller.setYAxis(getJoystickValue(1));

  // Joystick2
  controller.setRXAxis(getJoystickValue(2));
  controller.setRYAxis(getJoystickValue(3));

  // Triggers
  controller.setZAxis(get8BitAxis(4));
  controller.setRZAxis(get8BitAxis(5));
```

The treatment of our axes in the loop is fairly easy, we just capture the value using the functions explained before and use the result as the parameter for the methods of the controller library dedicated to update the values in the report. We continue with our buttons.

```
  // Buttons
```

```
  // Reads pins from 4 to 11
  for (int pin = 4; pin < 12; pin++)
  {
    controller.setButtonStatus(pin - 3, !digitalRead(pin));
  }
```

For the buttons we just iterate over the pins the buttons are wired to and rely on the *setButtonStatus(int button, int status)* from the library to update the report, the inner workings of the method are explained in the following section. Just after the regular buttons we handle the DPAD or hat switch.

```
// D-PAD
  // Reads pins from 0 to 3, the pressed direction is calculated by
conditionals.
  // Priority is right over left, then up then down in case opossing
directions are pressed.
  // Sides and the posible combinations with up and down are checked first,
then if neither right or left are pressed up alone and down alone are
checked for.
  // Direction switch expected connections are
  //   RIGHT --> Pin 0
  //   UP    --> Pin 1
  //   DOWN  --> Pin 2
  //   LEFT  --> Pin 3
  if (!digitalRead(0))
  { // RIGHT
    if (!digitalRead(1))
    { // RIGHT + UP
      controller.setDPAD(DPAD_UP_RIGHT);
    }
    else if (!digitalRead(2))
    { // RIGHT + DOWN
      controller.setDPAD(DPAD_DOWN_RIGHT);
    }
    else
    { // RIGHT alone
      controller.setDPAD(DPAD_RIGHT);
    }
  }
  else if (!digitalRead(3))
  { // LEFT
    if (!digitalRead(1))
    { // LEFT + UP
      controller.setDPAD(DPAD_UP_LEFT);
    }
    else if (!digitalRead(2))
    { // LEFT + DOWN
      controller.setDPAD(DPAD_DOWN_LEFT);
    }
    else
```

```
    { // LEFT alone
      controller.setDPAD(DPAD_LEFT);
    }
  }
  else if (!digitalRead(1))
  { // UP alone
    controller.setDPAD(DPAD_UP);
  }
  else if (!digitalRead(2))
  { // DOWN alone
    controller.setDPAD(DPAD_DOWN);
  }
  else
  { // No direction is pressed - Centered DPAD
    controller.setDPAD(DPAD_CENTERED);
  }
```

The concept of DPAD or hat switch is quite open on its physical implementation, it can either be a single device that mechanically allows for only a single direction to pressed at once or it is implemented, as in the case of our prototype, as 4 separate buttons that may be pressed individually. For this reason, we have to stablish a priority for the direction and use some logic in our code to decide which direction is sent when contradicting switches are pressed. We decided to prioritise right over left and up over down, and we check first the horizontal over vertical directions. This is done by the use of nested conditionals as stated above. As of now it is done as part of a loop, but in the case of having many DPADs or hat switches we could extract a function that taking the corresponding pins as parameters returns the value that should be included in the report.

```
  controller.SendReport();
}
```

To finalise our loop iteration, we call the function of our library to finally sent the report with the updated values.

Additionally, after this initial design was implemented and tested it was decided to repurpose the debugging LED to represent the status of the bit precision jumper, for this purpose the following line was added just after the obtention of its value each iteration.

```
  digitalWrite(13, !BIT_JUMPER);
```

Chapter 5.2.2 - The HID-Controller library implementation

In this section we elaborate on the implementation of our controller library, the basic design and purpose was already commented in previous sections including the split in the header, *.h*, and code, *.cpp*, files. We start by explaining the contents of the header file.

We start with the inclusions of the basic Arduino core libraries; they are required for the most basic functionalities of the code to work properly such as the use of the pin initialization or basic functions of the programming language, and then we also include the HID library, as while a core library provided by Arduino to ease the work of developers, it is not included with *Arduino.h* as it is not that common to use.

```
#include <Arduino.h>
#include "HID.h"
```

We continue with some useful definitions to make some parts of our code easier to understand, in this case instead of just using the numbers corresponding to a DPAD direction wherever they are needed we define names for each of them making the programming of the DPAD logic easier.

```
#define DPAD_CENTERED 0
#define DPAD_UP 1
#define DPAD_UP_RIGHT 2
#define DPAD_RIGHT 3
#define DPAD_DOWN_RIGHT 4
#define DPAD_DOWN 5
#define DPAD_DOWN_LEFT 6
#define DPAD_LEFT 7
#define DPAD_UP_LEFT 8
```

We now find the initialization of our variable containing the descriptor for our report, as recommendation by Arduino in their instructions for Pluggable HID we define it as part of the programmable memory by the use of the modifier *PROGMEM*. The descriptor will only be used once during the initialization of the device and later becomes irrelevant, so we save some of the faster SRAM with this practice. The way the descriptor is created is by placing each byte word after byte word to represent the descriptor that was designed for our device.

```
 static const uint8_t hidDescriptor[] PROGMEM = {

    //Controller with 8 buttons, 4 16(10)bit axis, 2 8bit axis and a 2
DPADs (1 used)
    0x05, 0x01,                         // Usage Page (Generic Desktop)
    0x09, 0x04,                         // Usage (Joystick)
    0xa1, 0x01,                         // Collection (Application)
    0x85, 0x01,                         //   Report ID
    // 8 Buttons
    0x05, 0x09,                         //   Usage Page (Button)
    0x19, 0x01,                         //   Usage Minimum (Button 1)
    0x29, 0x08,                         //   Usage Maximum (Button 8)
    0x15, 0x00,                         //   Logical Minimum (0)
    0x25, 0x01,                         //   Logical Maximum (1)
    0x75, 0x01,                         //   Report size (1)
    0x95, 0x08,                         //   Report Count (8)
    0x81, 0x02,                         //   Input (Data,Var,Abs)
    // 4 16bit(10) Axis
    0x05, 0x01,                         //   Usage Page (Generic Desktop)
    0xa1, 0x00,                         //   Collection (Physical)
    0x09, 0x30,                         //     Usage (X)
    0x09, 0x31,                         //     Usage (Y)
    0x09, 0x33,                         //     Usage (Rx)
    0x09, 0x34,                         //     Usage (Ry)
    0x15, 0x00,                         //     Logical Minimum (0)
    0x27, 0xFF, 0xFF, 0x00, 0x00,       //     Logical Maximum (65534)
    0x75, 0x10,                         //     Report size (16)
    0x95, 0x04,                         //     Report Count (4)
    0x81, 0x02,                         //     Input (Data,Var,Abs)
```

```
    // 2 8bit Axis
    0x09, 0x32,                    //     Usage (Z)
    0x09, 0x35,                    //     Usage (Rz)
    0x15, 0x00,                    //     Logical Minimum (0)
    0x26, 0xFF, 0x00,              //     Logical Maximum (255)
    0x75, 0x08,                    //     Report size (8)
    0x95, 0x02,                    //     Report Count (2)
    0x81, 0x02,                    //     Input (Data,Var,Abs)
    0xc0,                          //   End Collection
    // 2 DPADs (1 used)
    0x05, 0x01,                    //   Usage Page (Generic Desktop)
    0x09, 0x39,                    //   Usage (Hat switch)
    0x09, 0x39,                    //   Usage (Hat switch)
    0x15, 0x01,                    //   Logical Minimum (1)
    0x25, 0x08,                    //   Logical Maximum (8)
    0x75, 0x04,                    //   Report size (4)
    0x95, 0x02,                     //   Report Count (2). While only
1 report is used, this is necessary to match the bits position
    0x81, 0x02,                    //   Input (Data,Var,Abs)
    0xc0                           // End Collection
};
```

Then we arrive to a truly relevant part of the development, defining a way for the data of our report to be structured according to our descriptor for the report.

```
typedef union  {
    // 8 Buttons, 4 10bit axis, 2 8bit axis, 1 DPAD
    struct {
        uint8_t buttons;

        uint16_t xAxis;
        uint16_t yAxis;

        uint16_t rxAxis;
        uint16_t ryAxis;

        uint8_t zAxis;
        uint8_t rzAxis;

        uint8_t dPad;
    };
} HID_Report;
```

We define a union containing a struct. Having a union in C means we are allocating continuous memory for all the fields we allocate inside while what we really want to define is the struct for all our data, we just want to keep it from fragmenting so it can be passed to the *SendReport* function of the HID library. The struct by itself does not warrant our values are stored continuously, storing it inside a union does. As for the contents we are allocating a full byte for our buttons, a bit for each, 2 bytes for each 16 bit axis, 1 bit for each 8 bit axis and finally a full byte for our single DPAD and the second unusable DPAD. We would use the first 4 bits for the existing DPAD and the last 4 for the theoretical one. As there is no type in Arduino for a 4 bit integer and using 4 single bit variables for the DPAD would add unnecessary complexity to the

library and increase the number of operations the compromise of declaring a non-usable DPAD was accepted. This kind of compromises to match the bit positions can be found in some devices on the market from first grade manufacturers, for example, the Xbox One Controller states to be reporting 16 buttons while it actually presents 11 physical buttons for the user to press.

Finally we declare the Controller class that we use in our *.ino* file and implement its methods in the *.cpp* one.

```
class Controller {
public:
    Controller(void);
    //Report and initialization
    void SendReport(void);
    void initialize(void);
    void zeroEverything(void);
    //Button handling
    void setButtonStatus(uint8_t button, bool status);
    //Axis Handling
    void setXAxis(uint16_t value);
    void setYAxis(uint16_t value);

    void setRXAxis(uint16_t value);
    void setRYAxis(uint16_t value);

    void setZAxis(uint8_t value);
    void setRZAxis(uint8_t value);
    //DPAD Handling
    void setDPAD(uint8_t value);
protected:
    HID_Report report;
};
```

This are only the method declarations and the declaration of the report variable using the type defined for our report before. We detailed the implementation of each method in the *.cpp* file where we first are to include the header file so we can correctly reference the methods we are implementing.

```
#include "HID_Controller.h"

Controller::Controller(void)
{
    static HIDSubDescriptor node(hidDescriptor, sizeof(hidDescriptor));
    HID().AppendDescriptor(&node);
}
```

The first method is the constructor for the Controller object, it creates an object of the *HIDSubDescriptor* class from the HID library using the descriptor we defined in the header and finally by using the *AppendDescriptor* method of the library sets the USB connection to be recognized by the host system as the intended HID device.

```
void Controller::SendReport(void)
{
    HID().SendReport(0x01, &report, sizeof(report));
```

```
}
```

The *SendReport* method just calls with the appropriate parameters the *SendReport* method of the HID library, the parameters are the ID of our report as indicated in the descriptor, 1, the reference to the memory location where out report is stored in a continuous fashion due to the use of the struct nested inside a union, and the number of bits that have to be read from that point onwards.

```
void Controller::initialize(void)
{
    zeroEverything();
    SendReport();
}

void Controller::zeroEverything(void)
{
    report.buttons = 0;
    setXAxis(0);
    setYAxis(0);
    setRXAxis(0);
    setRYAxis(0);
    setZAxis(0);
    setRZAxis(0);
    setDPAD(0);
}
```

The *initiliaze* and *zeroEverything* methods are commented together due to its relation. When first sending the report of a HID device it is recommended to send all the values set to zero to prevent casualties like buttons being incorrectly pressed, so as initialization process for our controller we decided to use a method to first set every value of the report to zero and then instantly send the report for the first time without updating any of the values to its current state. As a curiosity, when not using this initialization method during testing pressing a button while connecting the board to the computer could result in the HID device part of the board becoming unresponsive, requiring reconnecting or reprograming it before it could function again.

```
void Controller::setButtonStatus(uint8_t button, bool status)
{
    bitWrite(report.buttons, button - 1, status);
}
```

The way our buttons are represented in the report is with the status of bit each inside the byte allocated for then in the report.buttons variable of the *HID_Report* type. For this reason, the way their status is updated in the report is by collecting its status and then using the *bitWrite* native function to update specifically that bit position in the variable.

```
void Controller::setXAxis(uint16_t value)
{
    report.xAxis = value;
}

void Controller::setYAxis(uint16_t value)
{
```

```
    report.yAxis = value;
}

void Controller::setRXAxis(uint16_t value)
{
    report.rxAxis = value;
}

void Controller::setRYAxis(uint16_t value)
{
    report.ryAxis = value;
}

void Controller::setZAxis(uint8_t value)
{
    report.zAxis = value;
}

void Controller::setRZAxis(uint8_t value)
{
    report.rzAxis = value;
}
```

The axes are easier than the buttons to update in the report, as their report size matches the size of their corresponding variables, we are able to just write the precision translated values without any bitwise treatment.

```
void Controller::setDPAD(uint8_t value)
{
    report.dPad = value << 4;
}
```

Finally, we have the method to update the value of our DPAD, in this case we are to write the value in the first 4 bits of the allocated byte for both the intended DPAD and the additional unusable one. If we were to use multiple DPAD we would have to modify this method, so it only sets those 4 bits while leaving the rest unmodified, the same applies to any other method created to set the values of any control sharing variable with others, only their corresponding bits are to be modified.

Chapter 6 - Test prototype design and implementation

In this chapter we comment on the established required elements that should be present on our testing prototype and the process of how it came to live starting with the basic schematics of each components connectivity and finally the end result.

Chapter 6.1 - Required human interaction elements

It was decided for our software design that we would make ample use of the board capacities when it comes to controls connectivity, we have available 14 digital pins and 6 analog ones, and as such we settled on 8 regular buttons, a DPAD taking 4 digital pins and 6 axes occupying the complete analog pin array leaving the pin number 12 for a jumper that allows us to select the axes bit precision and the pin 13 for our first debugging LED and then bit precision indicator. In order to accomplish this, it was decided to use the following elements:

- 10 regular buttons connected directly to the digital pins set as *INPUT_PULLUP*. This set would be split so 4 of then compose the DPAD or hat switch replacement and 6 for buttons.
- 2 Fully wired of our joysticks. Each joystick is able to report the value for 2 axes and button when every pin is properly connected to our board. These 2 joysticks account for a total of 4 axes, more specifically the high precision ones, and 2 buttons.
- Additional joystick with no button wiring. For the lower precision axes, it was decided to add a third joystick to the protype, in this case wiring only the pins dedicated to the two axes this joystick is able to report.
- Jumper for the bit precision.
- In the case of the LED in pin 13 we are referring to the on board LED, no wiring is required.

Chapter 6.2 - Electronic design and schematics

In this section we will first comment the wiring for each individual component in detail resulting in a minimal component of our system, the subsequent repetition of the patterns created to connect each element will then result in the complete prototype. Both the schematics and explanation graphs in this section have been generated with the online tool Tinkercad.

Chapter 6.2.1 - Minimal component connection schematics

Chapter 6.2.1.1 - Single button connection

In this case we opted for the pull-up variant for our buttons in which basically we use the internal resistor in the Arduino instead of adding our own components externally to the board which would increase the assembly complexity, space requirements and costs to any build. When assembling a switch circuit we would have 3 important points, the voltage source, ground, and then our reading point where we obtain the potential which we will interpret as HIGH when corresponding to the voltage of our source or LOW when it is 0 or virtually so. The way the voltage is altered is by the use of a switch to open or close parts of the circuit, where the switch is placed determines whether it will be a pull-up or pull-down switch circuit. Any configuration has to include a resistor to prevent excessive current that would damage the system.

We take as reference this explanation from the fifth revision of the practical guide for the Computer Electronics Technology subject [22], more specifically from its summary of the Arduino reference. For the case of the pull-up setting, we can see that by the use of a transistor the pin is internally connected to the 5V source passing through a 20K Ohms resistor, then our external switch or button is placed after the point where the potential is read. This means that when opened there will be no path from the 5V source to the ground, the potential is 5V and as such the value obtained is HIGH. When the switch is closed the 5 volts go directly to the ground meaning no potential and a value of LOW.

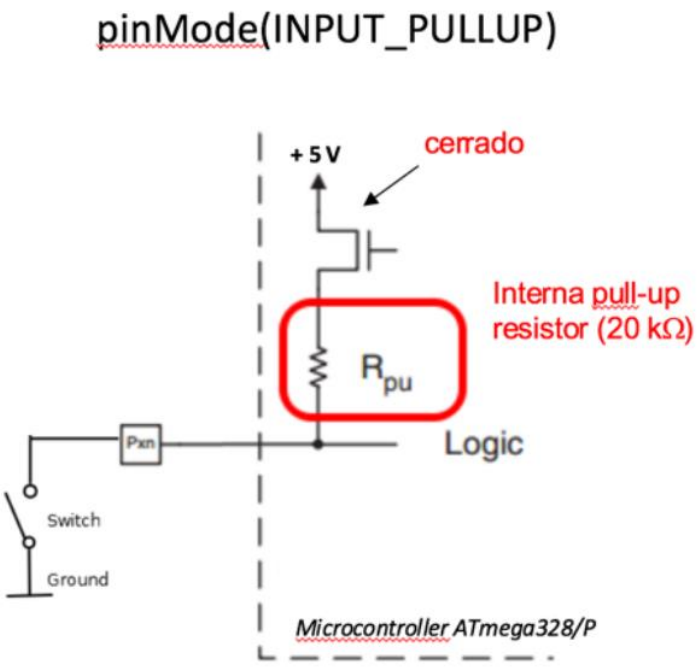

Figure 19 - Input pull-up connected switch as explained in the practical guide for TEC

The basic concept of how to connect a switch or button of the kind selected can be seen in the following schematic and graphic.

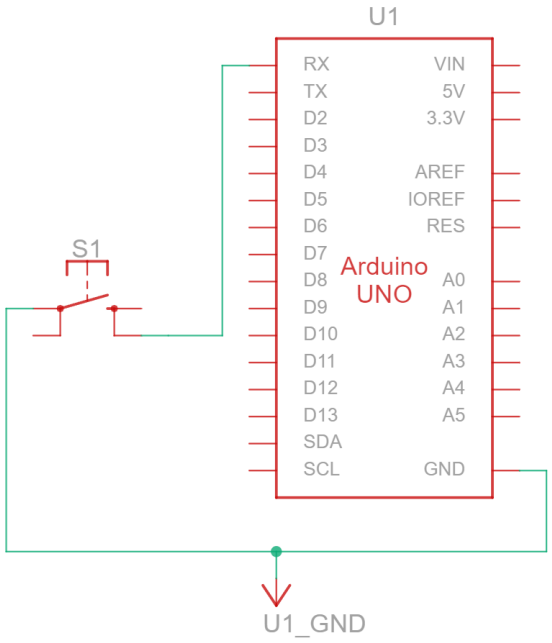

Figure 20 - Schematic for a button

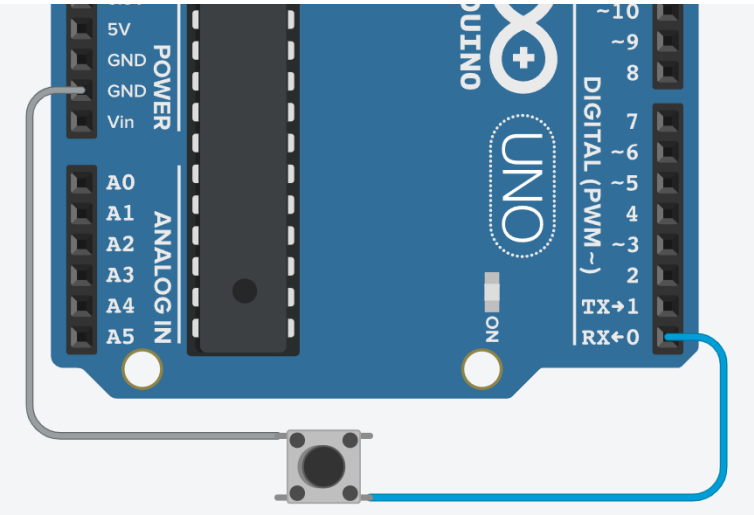

Figure 21 - Graphic of a button wiring

Chapter 6.2.1.2 - Fully wired joystick connection

For our prototype we are using joystick modules intended for Arduino that allow us to wire 2 analog axes and a pull-up style switch easily, they only require us to wire the 5V and ground pins and then we may wire the axes and the switch pin as needed. Any of these connections can be done with a single wire directly to the pins of our Arduino board. For our schematic and Tinkercad graph we had to make a compromise as the tool does not include joystick modules, instead they were replaced by 8 pins connectors where we may interpret each as the corresponding pin of our module:

1. GND. Ground.
2. +5V. 5 volts input.
3. VRx. Connector for the output of the potentiometer giving the value of the horizontal axis.
4. VRy. Connector for the output of the potentiometer giving the value of the vertical axis.
5. SW. Connector for the output of our switch.

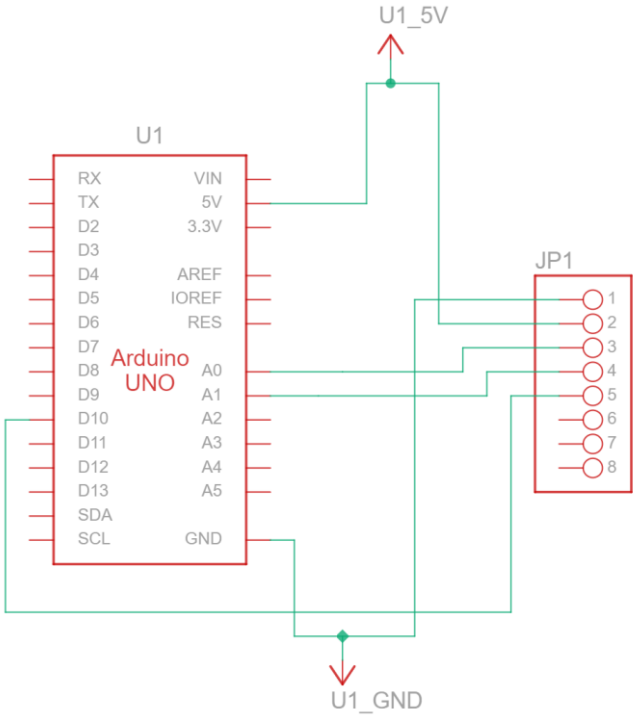

Figure 22 - Joystick wiring schematic

Our graph for the joystick connection does include a representation of the insides of a joystick module. While not realistic in the way the wires are connected to the 8 pin connector, many wires connected to the same pin, this is only a representation of in the PCB traces and the components soldered to it. We would have a 5V rail feeding our potentiometers and the ground rail to close the circuit, then we have two wires, in black and white coming out of their respective potentiometer outputs and connecting our VRx and VRy pins. The switch just has one end connected to ground and the other to the SW pin, the PCB just acts as a bridge to the pins while remaining just a switch with two ends.

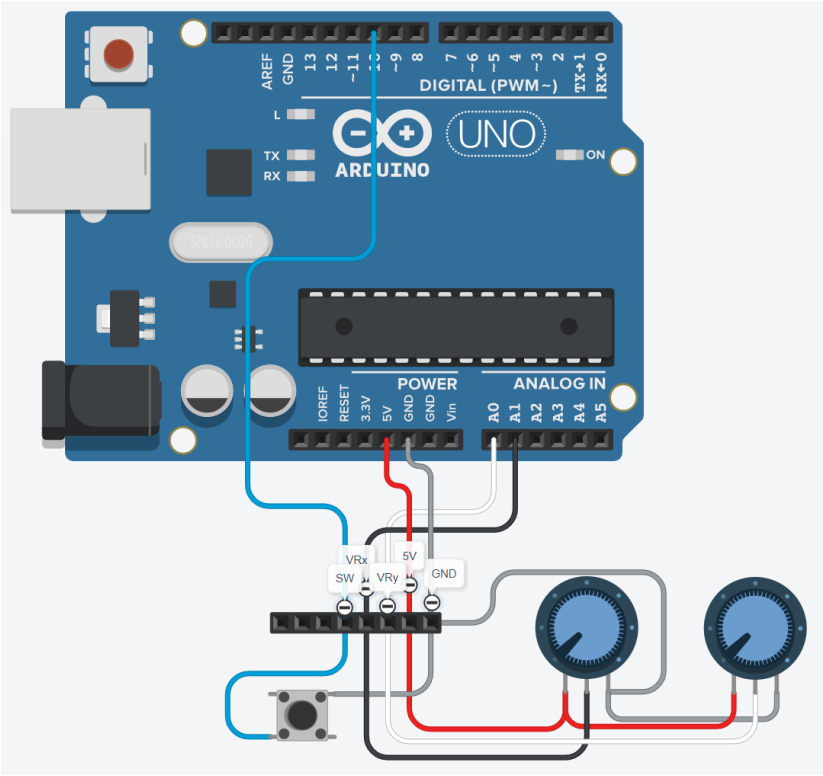

Figure 23 - Joystick graph with inner representation

Chapter 6.2.1.3 - Bit precision jumper

This is the most basic component of our prototype, it is functionally identical to the switches, but instead of being a permanent connection that can be opened or closed by pressing the button we are using the connection or not of a wire to simulate a position locking switch that would be used in a more elaborated build. We are simply connecting a wire to the pin 12 and then connecting it or not to ground to change the value.

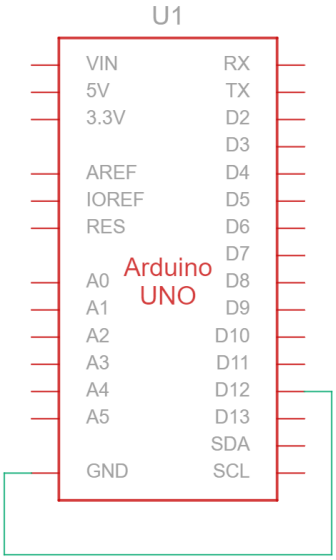

Figure 24 - Schematic for the BIT_JUMPER

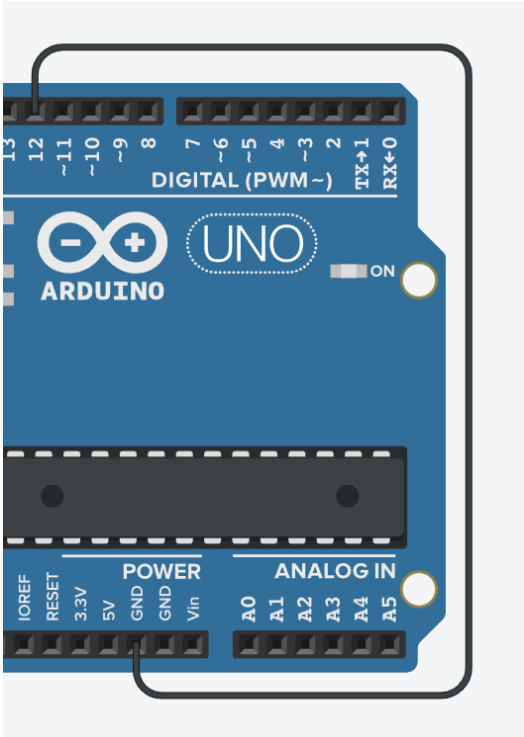

Figure 25 - BIT_JUMPER wiring graph

Chapter 6.2.2 - Full prototype schematic

This schematic is the result of putting together the knowledge of how to wire each individual part together with the use of a breadboard for prototype assemble. Firstly, two rails are established by connecting the 5V and the ground from the board to the intended rails of the breadboard and using some extra wires to extend those rails to the other side. The rails can now be used to obtain a 5V or ground connection at any needed place of the board, we just extend a wire from the rail to the transversal row where our component will be connected, the rows act as a direct connection between any components connected to it. In our prototype the rows are only used for series connection between a wire and a component such as a switch or the connectors to our joystick modules.

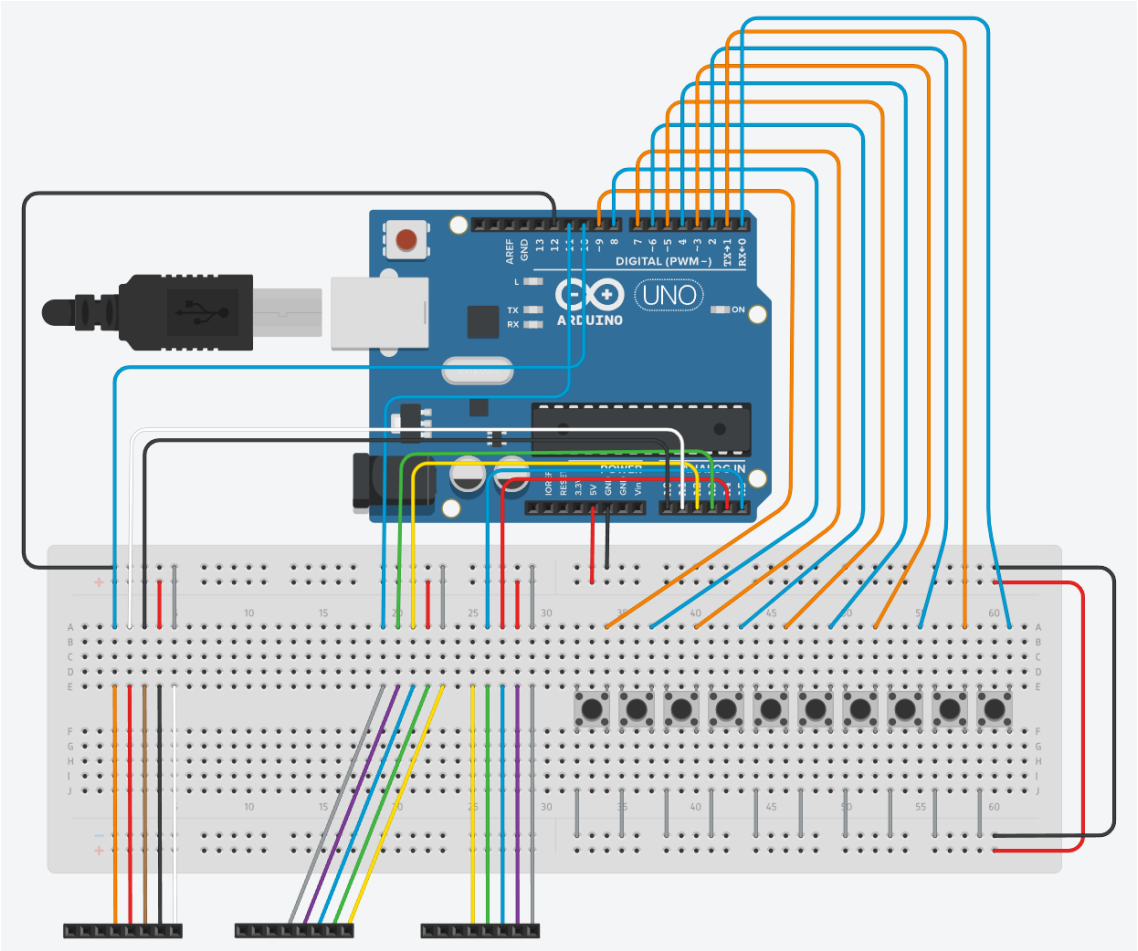

Figure 26 - Full prototype wiring graph

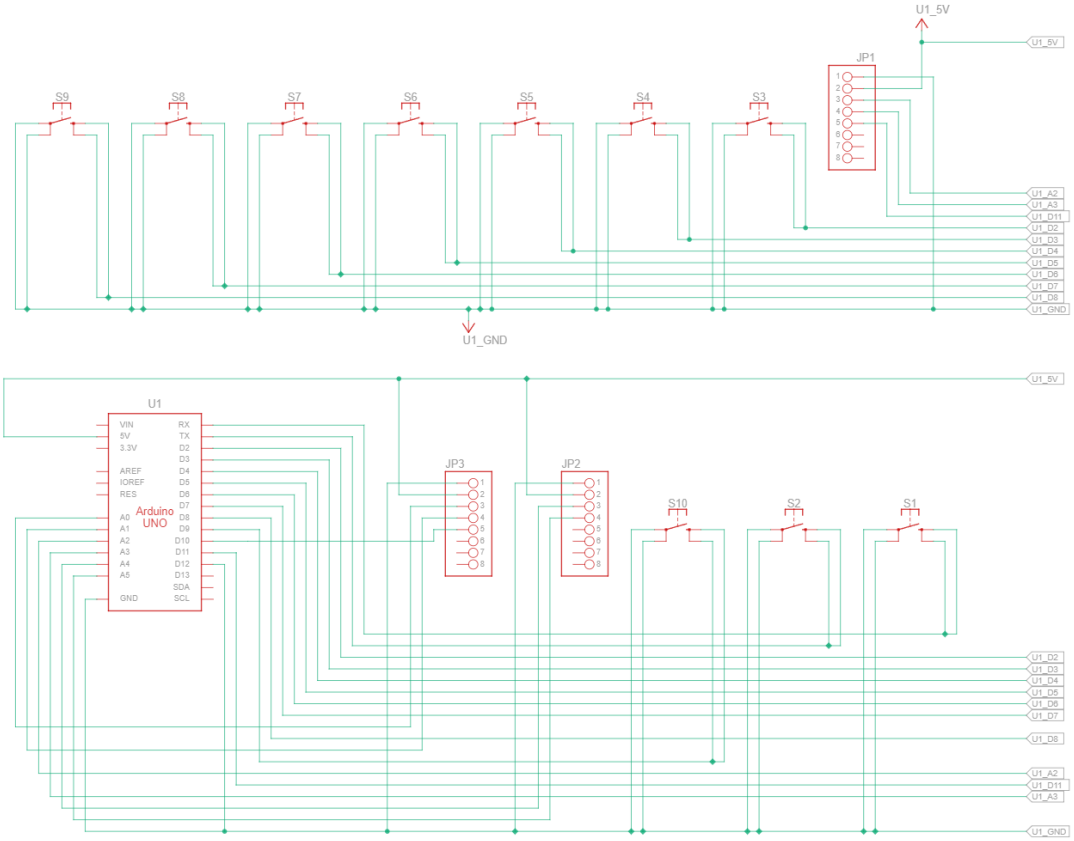

Figure 27 - Full prototype schematic

Chapter 6.3 - Prototype assembly

We proceed with the construction of the prototype, for this explanation of the process some steps are performed in stages and parts disassembled so the relevant ones can be appreciated without visual clutter.

Firstly we prepare the breadboard by wiring the aforementioned 5V and ground rails and connections to the breadboard rows. The wiring of this stage is done using the solderless rigid jumpers included in the electronics kit in order to make structurally resilient and low profile connections improving the ergonomics of the prototype to some degree. The buttons are also a low profile component so they are placed in the board according to the schematic.

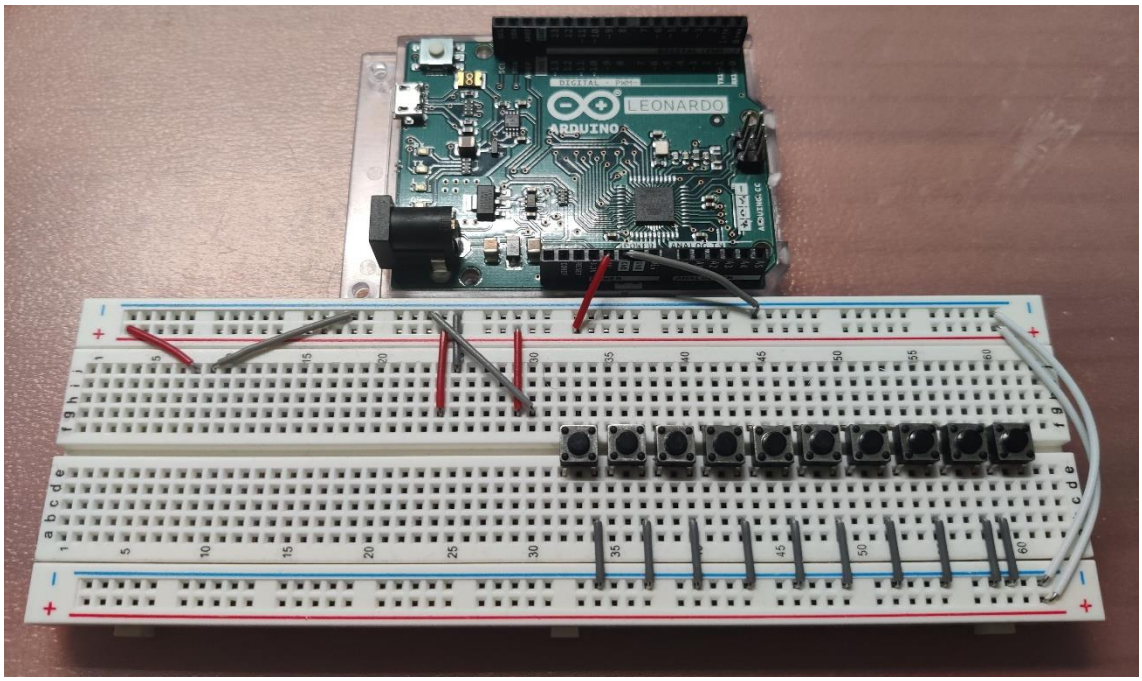

Figure 28 - Wiring of the 5V and ground rails, button placement

We now proceed to place the bit precision jumper, as stated before it is a simple wire that is connected to the pin 12 and might be connected or not to ground on the other end to act as a selector.

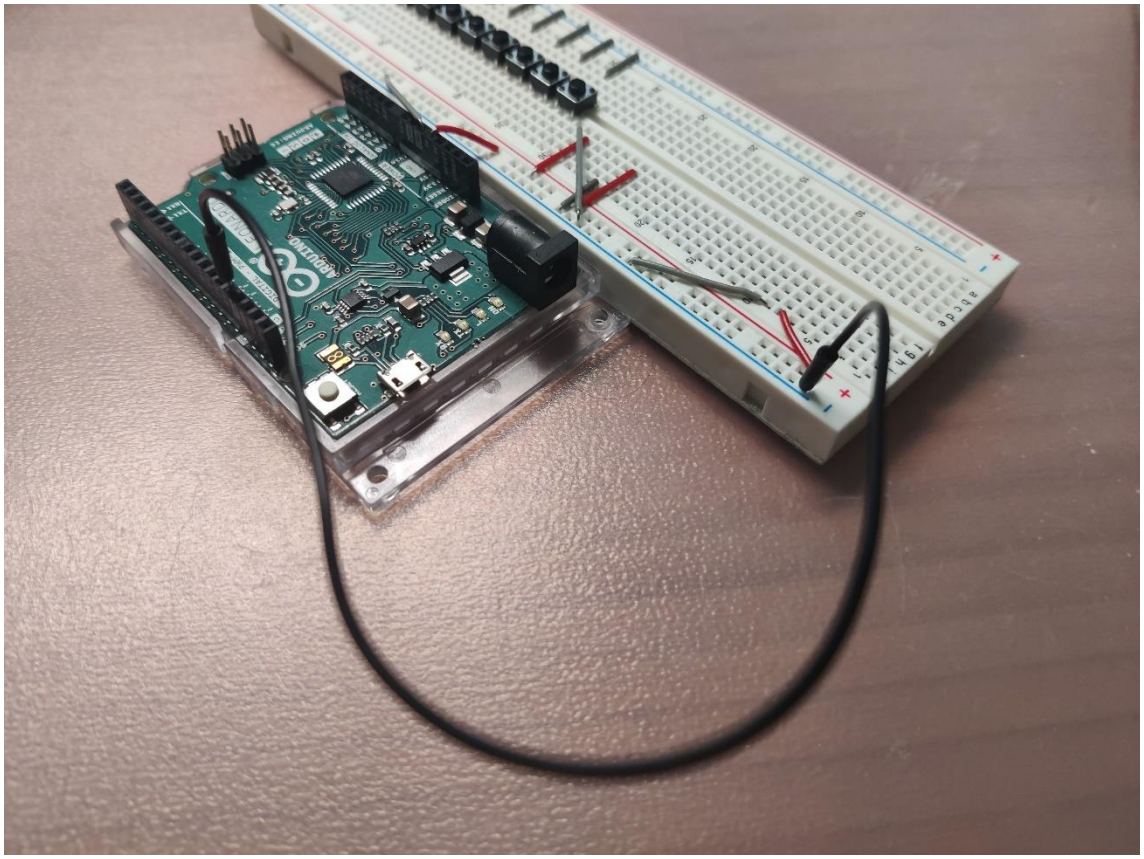

Figure 29 - Bit precision jumper detail

Then, we start with the breadboard mounted buttons wiring. They are already connected to the ground rails son only the wire from the corresponding transversal row to each digital pin is needed, we first see the detail of a single button connected to the pin 0, the right button of our DPAD, and then all of them together.

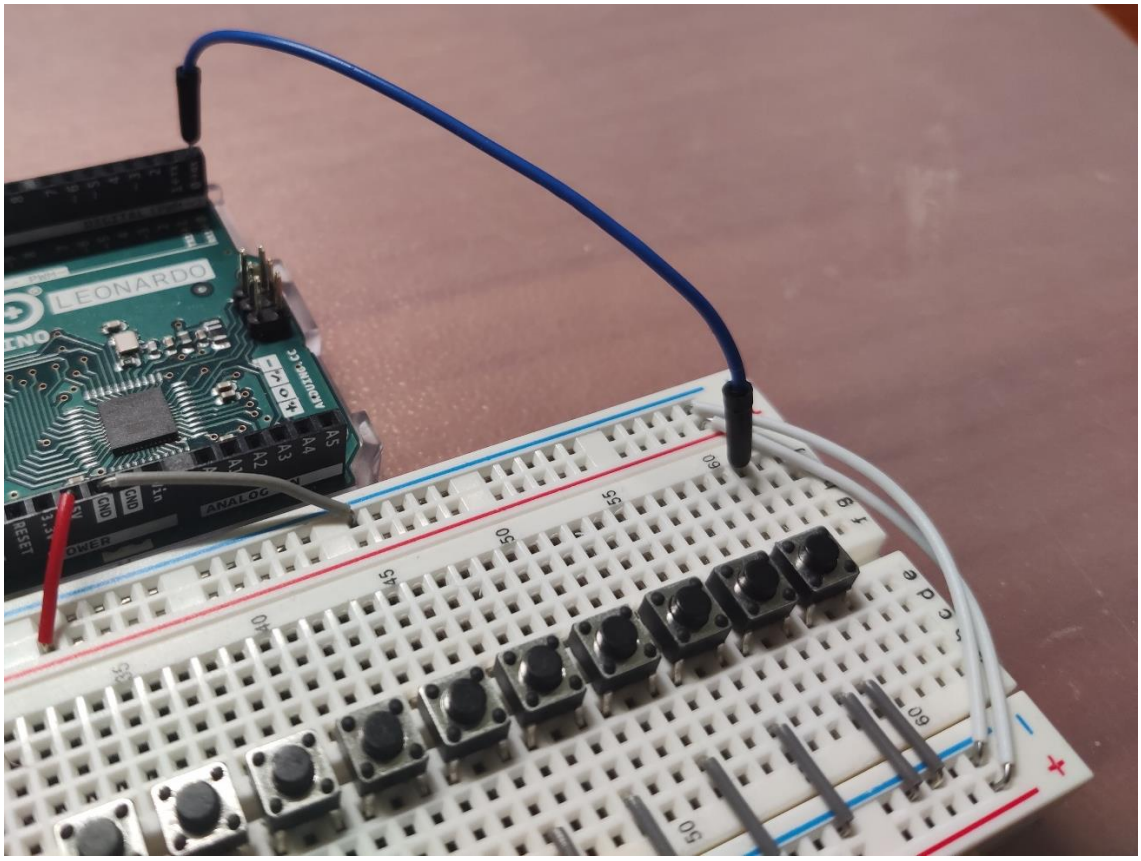

Figure 30 - Single button wiring detail

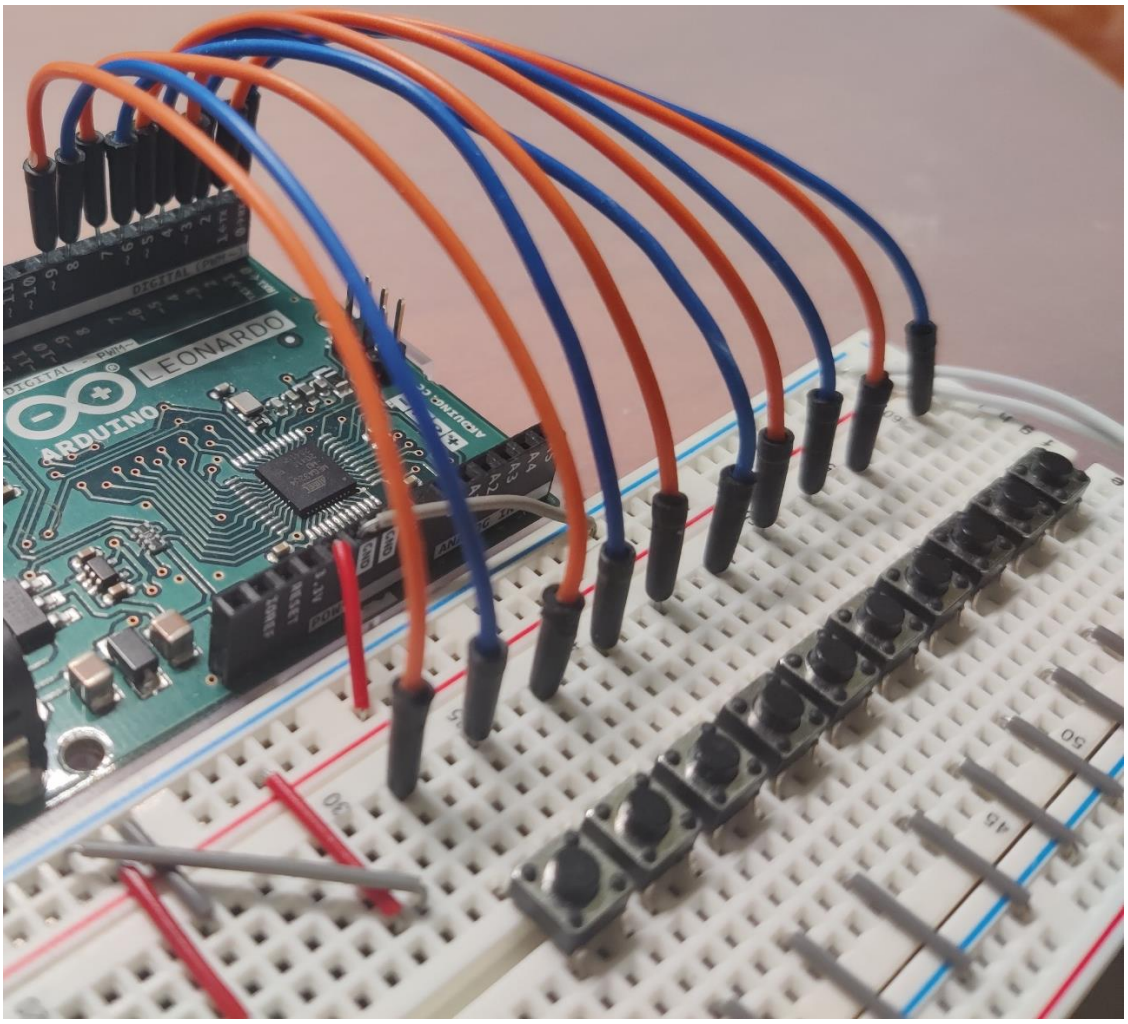

Figure 31 - Breadboard mounted button array

We continue with the connection of our joystick modules, first we appreciate the detail of the connectors present in the module and how they can be adapted to be used in our prototype with the Female-Male Dupont wires in groups of five to create a resistant but still flexible band.

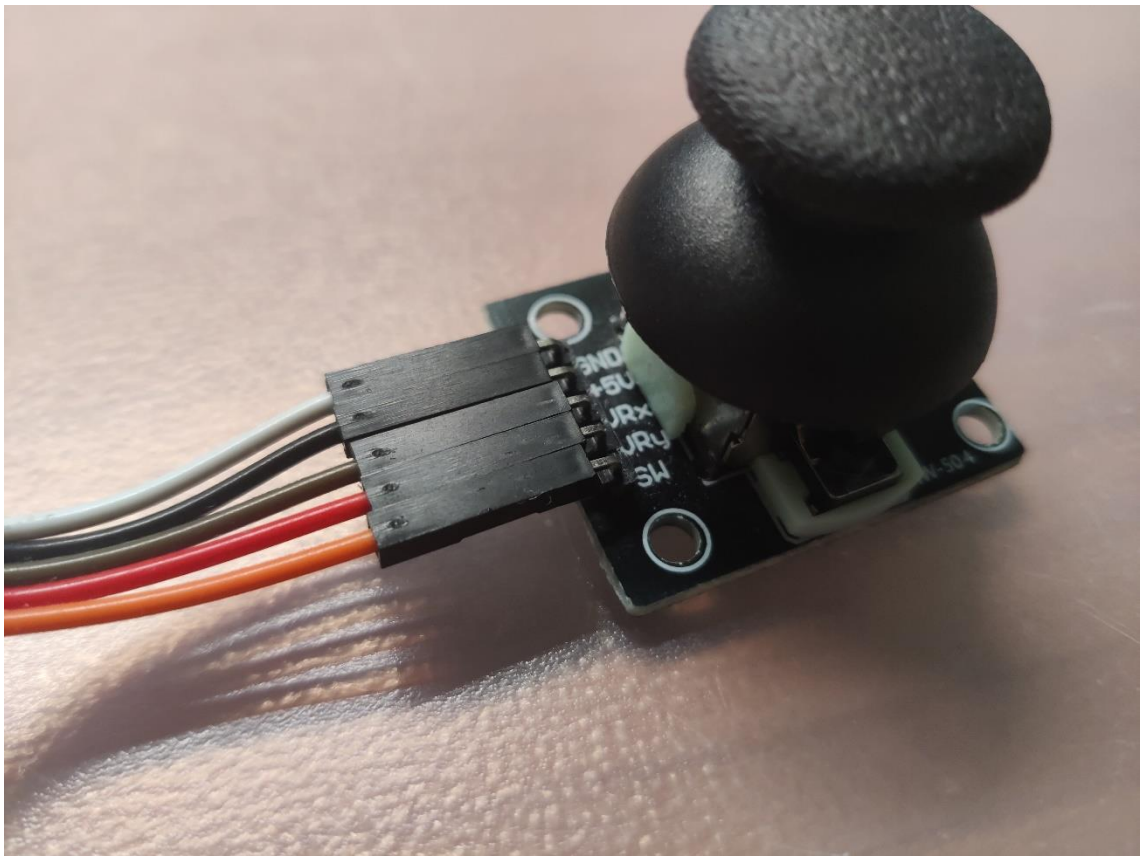

Figure 32 - Joystick module with wire-band

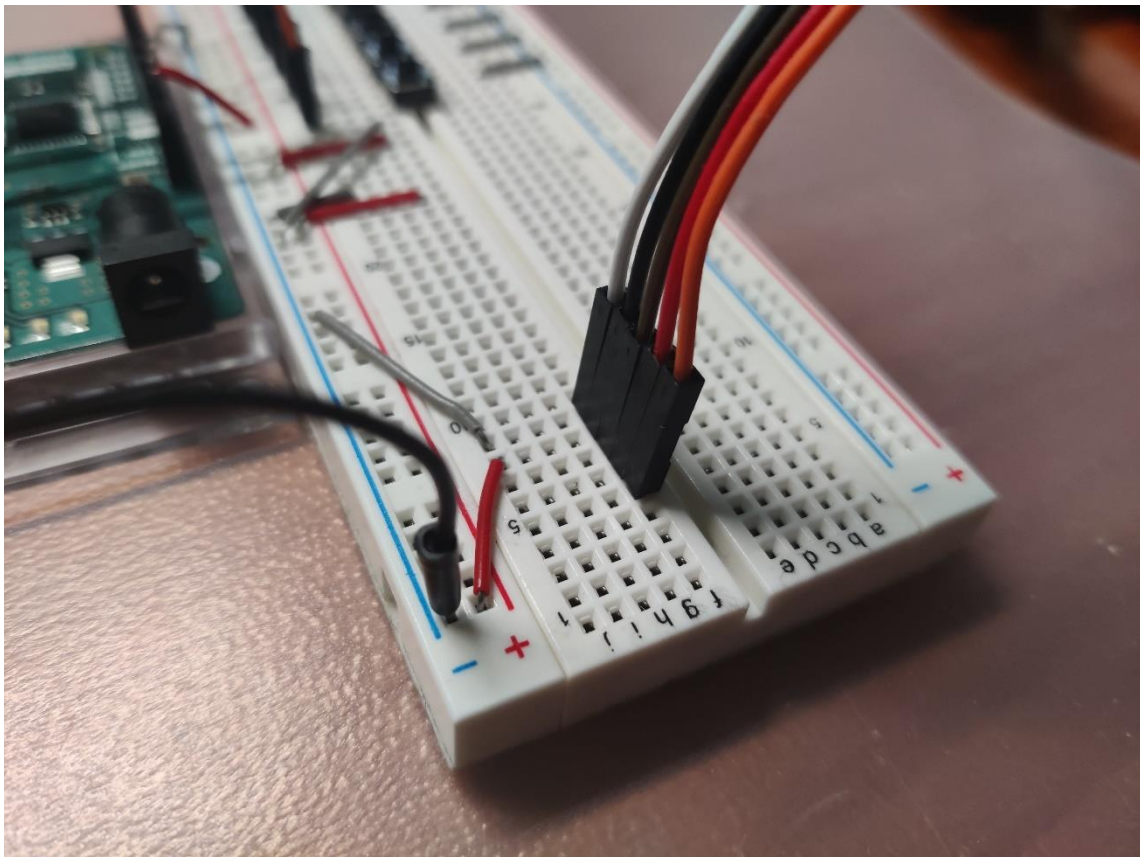

Figure 33 - wire-band connected to the breadboard

With this part of the assembly we already have a joystick module connected to the breadboard having all its connectors wired to five consecutive rows of the breadboard, one already connected to ground and another to the five volts, now we just have to wire the rows corresponding to our axes and the switch to their corresponding analog and digital pins.

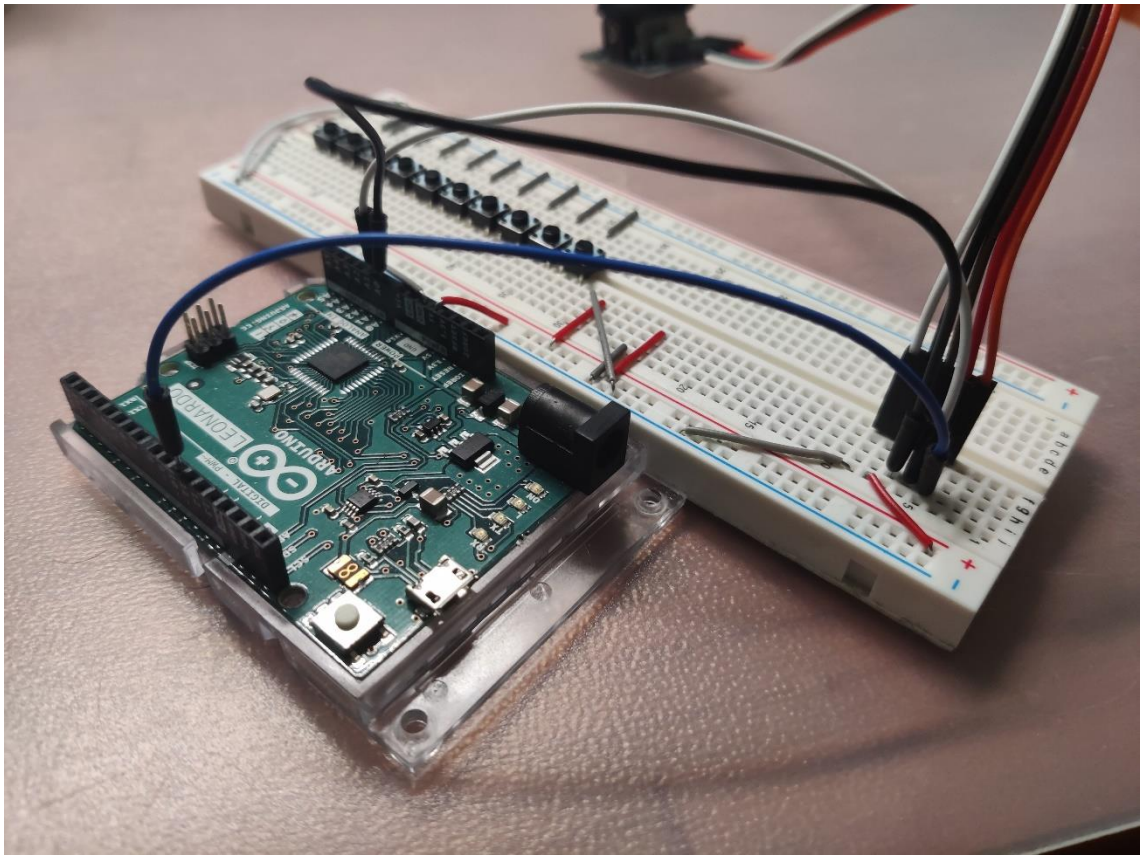

Figure 34 - Detail of a single joystick wiring

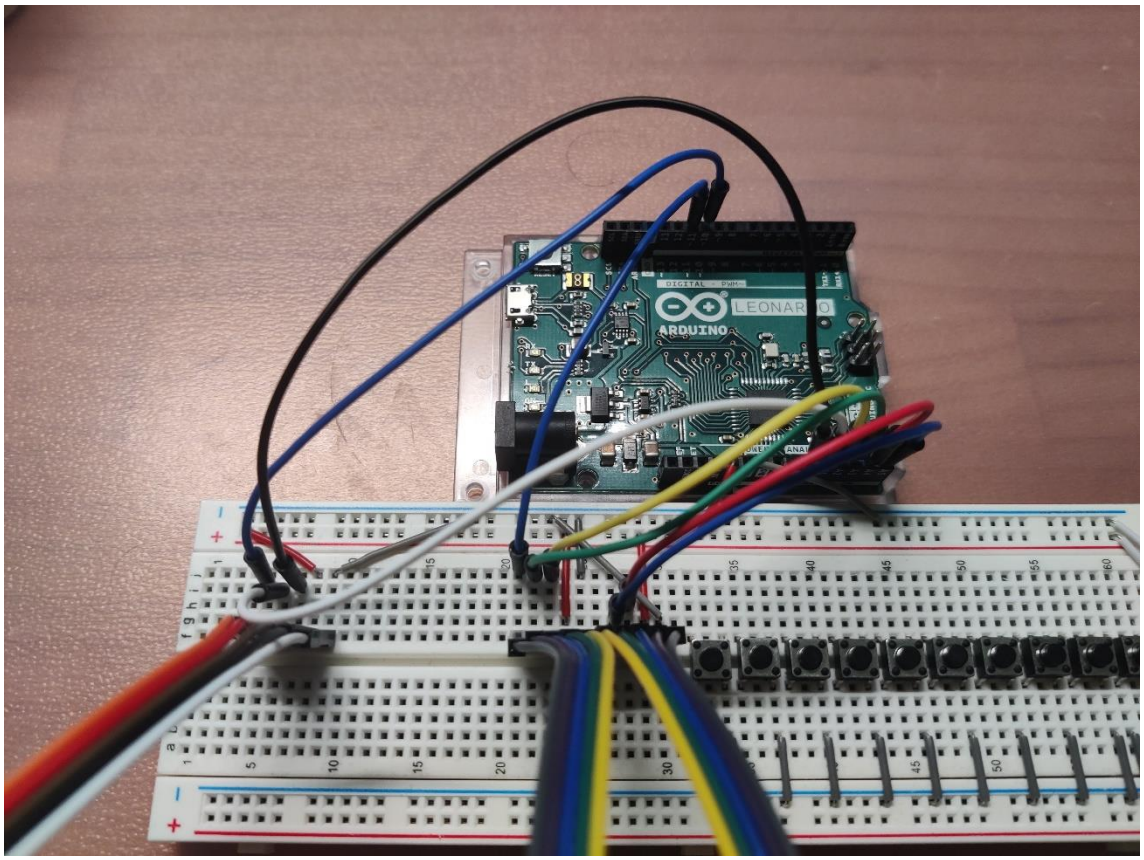

Figure 35 - Complete joysticks wiring

With the assembly of every individual part explained we can then perform the full assembly of the prototype and obtain the full result.

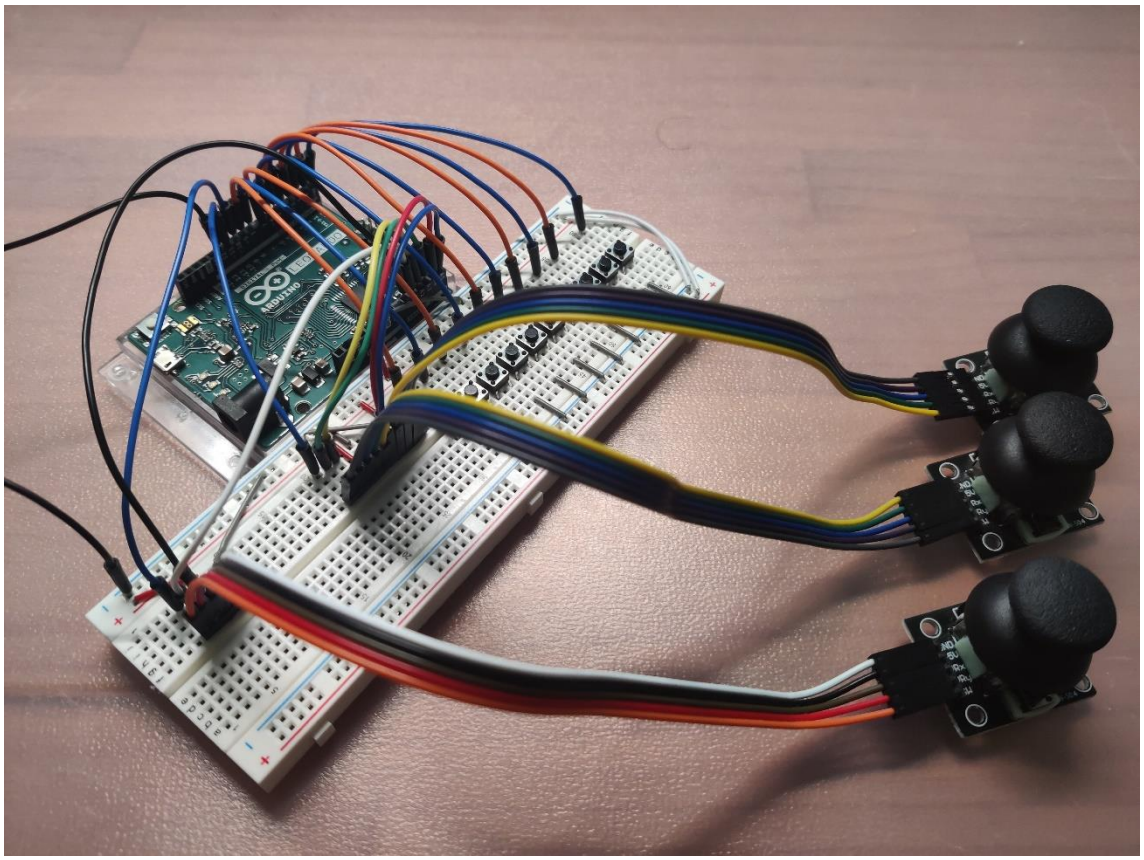

Figure 36 - Complete prototype assembly

The only element missing from this prototype in the pictures is a micro USB cable used to connect the board directly to the host computer, this cable does provide enough power to feed the system, meaning no additional power supply is necessary for our instance.

Chapter 7 - Acceptance testing for the prototype

The purpose of the prototype was from the beginning serving as a demonstration that the developed software accomplished the required functionality of translating the inputs in physical controls to a host system using the HID protocol, that can be explained in many requirements the prototype has to comply in order to be accepted, some implied by the standard and others are just desirable characteristics of a device of this kind or were included as required functionalities.

Chapter 7.1 - Compliance requirements for the prototype

- Standard derived requirements:
  - The device is powered and recognized on connection to a computer host.
  - No proprietary driver installation is required, the device is automatically setup as a gaming device with the specified controls.
    - This includes both the computer recognizing the existence of the controls and the physical controls being translated into values in the host computer.
- Desirable characteristics:
  - Data from every control is sent simultaneously. Some devices may not send every button pressing when many are pressed at the same time, while not expected in this case due to the way the code was designed, it could be possible to reach the limits of the controller, which would be invalidating the device functioning.
  - No appreciable input lag for the controls. If the controls were delayed it would result in great discomfort when using the controls for its intended use, latency when controlling any kind of game and specially flight simulators can result in miss inputs when trying to perform actions in a temporized manner and motion sickness when the movement is controlled in three dimensions and your actions do not correspond with what the screen shows.
- Additional features to be validated:
  - Values of the 16 bit precision axes are altered by the use of the bit precision jumper.

Chapter 7.2 - Testing methodology and results

For testing the device two stages were defined firstly verifying the inputs through the system settings available in Windows and then a real use case to check the performance of the prototype.

Chapter 7.2.1 - Artificial testing through system settings

This stage is to be performed in a system where the device has never been connected to or traces from that connection have been erased. In order to verify the requirement of not needing any proprietary driver for our device is HID compliant the system should recognize and set up the device on its own without user input. We connect the device to the system and obtain the following notifications, first the configuration one and then the installation confirmation.

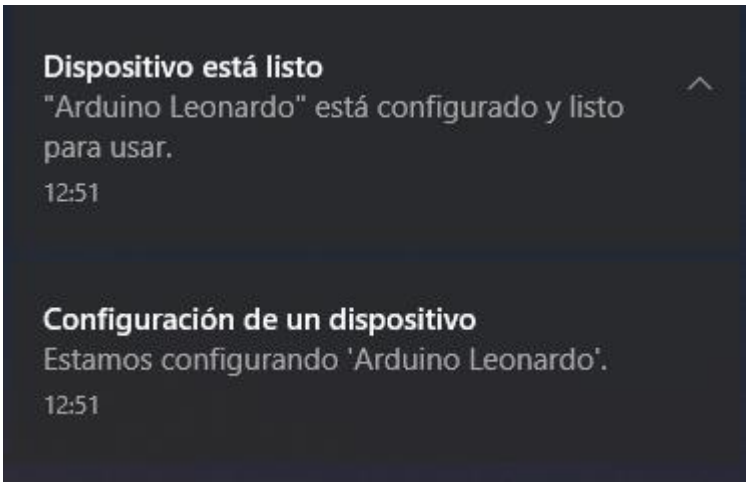

Figure 37 - Device installation notification

We can now go to the "Device and printers" section in the system settings and check that our device was recognized as a gaming device.

Figure 38 - Leonardo recognized as a gaming device

We can now enter the configuration for the gaming device and check the controls for our device are recognized, this image was taken with every control in its neutral position before calibrating the axes.

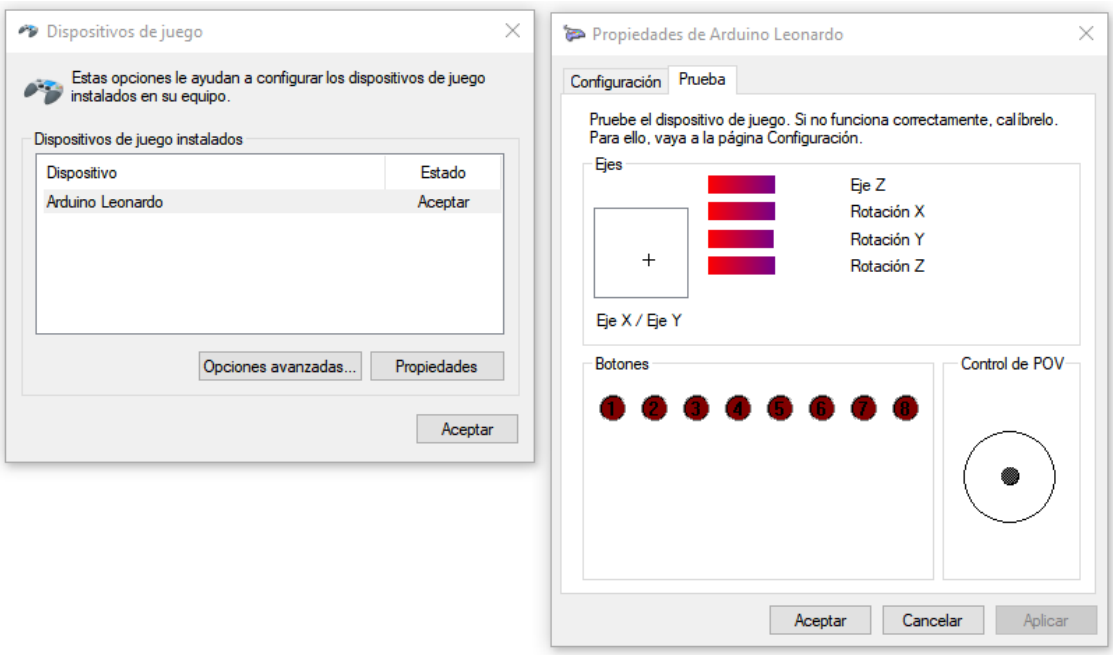

Figure 39 - Gaming devices setting for the prototype

We now proceed to calibrate the axes through the assistant, during this process we can also see the unprocessed data received by the computer, which allows us to verify the data of our axes is in the correct range and that the bit precision jumper functionality is working as intended. The images are taking after following the instructions for the calibration of the X and Y axes, those connected to our first joystick in pins 0 and 1. The first image represent the neutral position in 8 bit mode, the translation to 16 bit is correct and the value is stable.

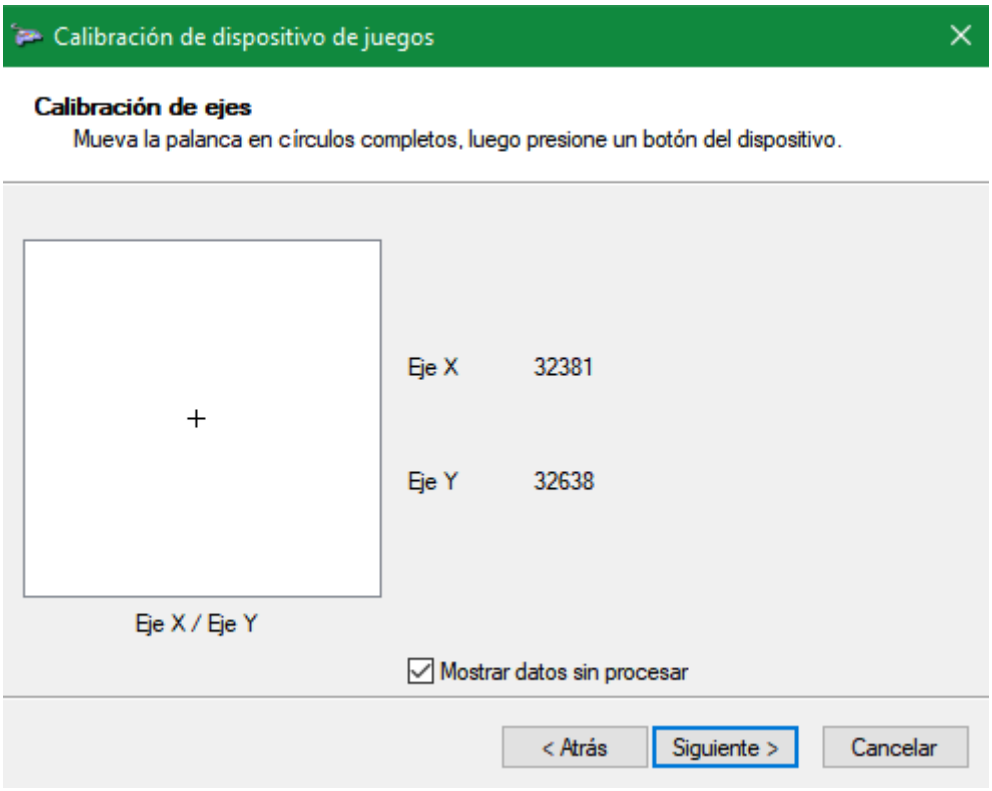

Figure 40 - X and Y axes, neutral position, 8 bit mode

Without touching the joystick, we set the jumper to 10 bit mode to take another capture, while it cannot be appreciated in a still picture, the values now jump between close ones, the step between one position and the next is closer and it is correctly represented.

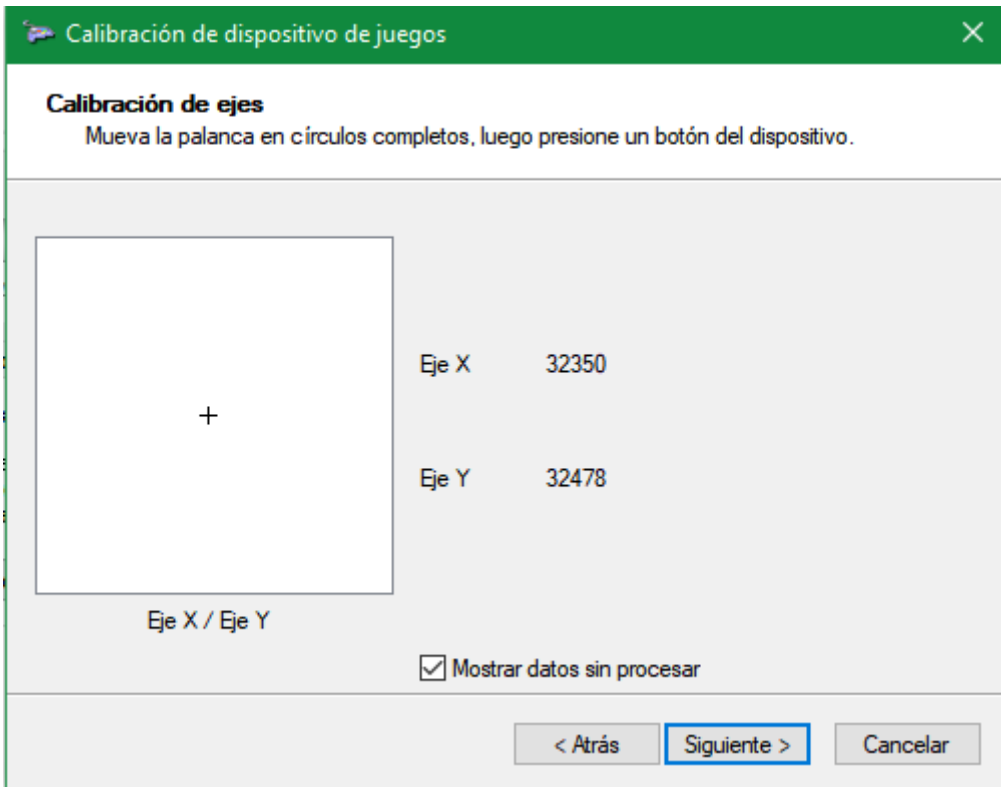

Figure 41 - X and Y axes, neutral position, 16 bit mode

The values at the corners for the maximum and minimum are also verified to see the differences between modes and also validating the correct translation at the edges. Fluctuation in 10 bit mode

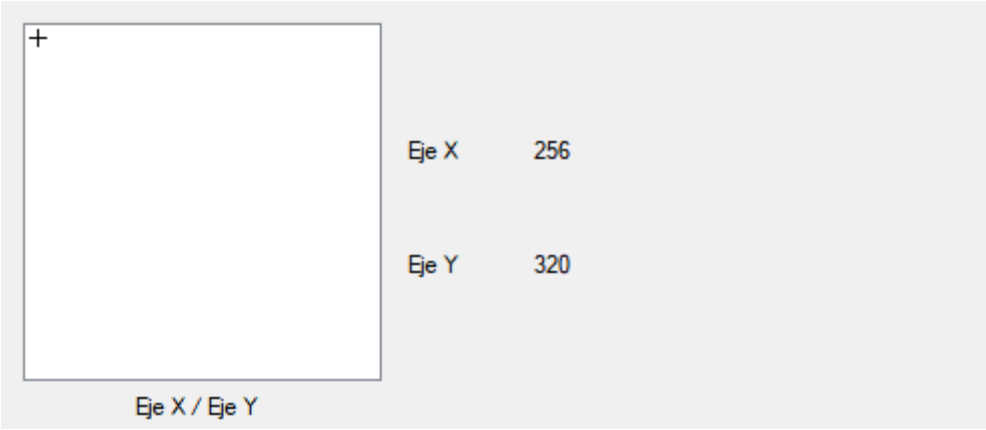

Figure 42 - Minimum X and Y values in 10 bit mode

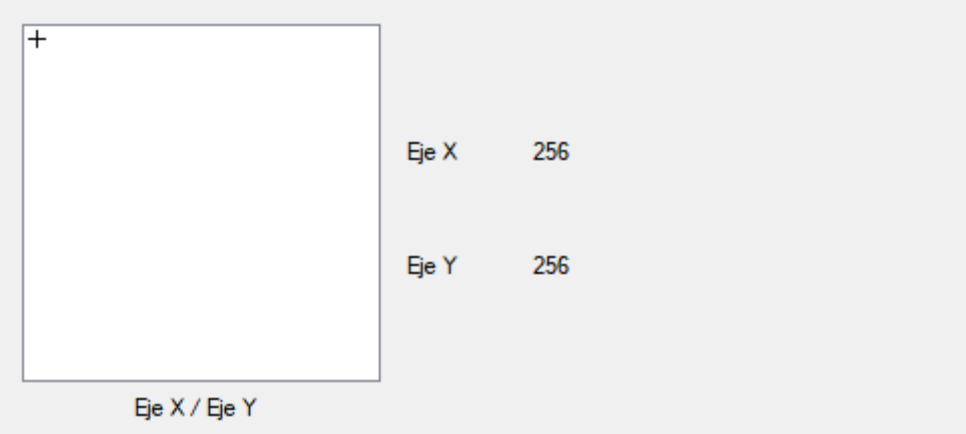

Figure 43 - Minimum X and Y values in 8 bit mode

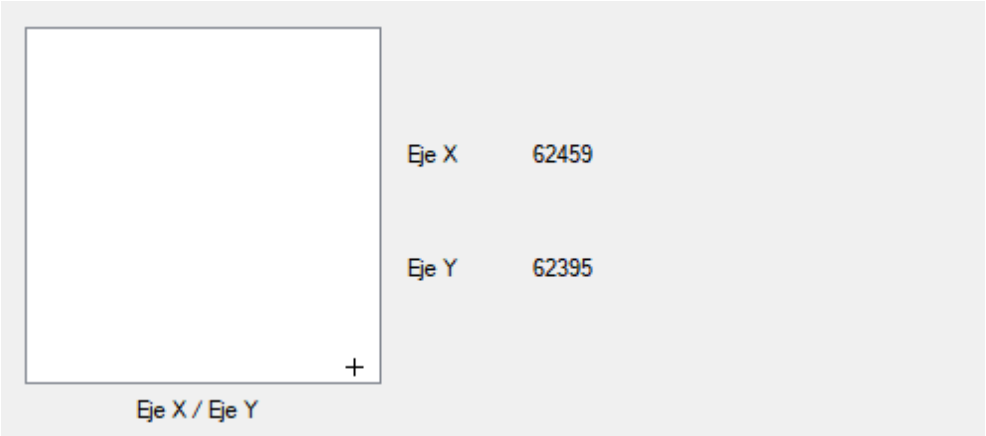

Figure 44 - Maximum X and Y values in 10 bit mode

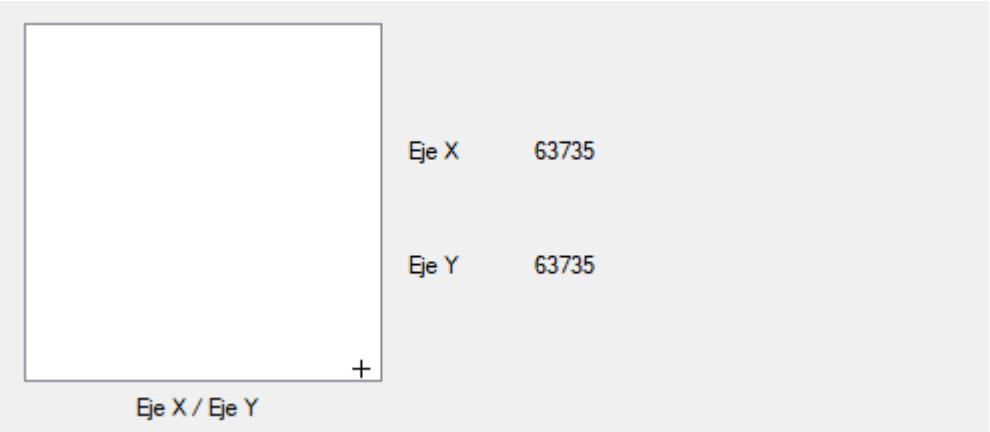

Figure 45 - Maximum X and Y values in 8 bit mode

At the maximum values, the results were found to be affected by the position of the wires and each time the joystick was moved to that position from neutral. This is caused by the low gauge wires and low fidelity potentiometers used in the modules. For the case of the bit precision jumper, it was verified it works for this pair of axes and that the LED, pin 13, does indeed switch on and off as intended when changing the mode.

We verify the rest of the axes, the other 16 bit axes, RX and RY present the same behavior as X and Y, then there are the 8 bit axes, Z and RZ. The 8 bit axes report values only from 0 to 255 and are not affected to the bit precision jumper, this was verified during testing. We use the Z axis as example.

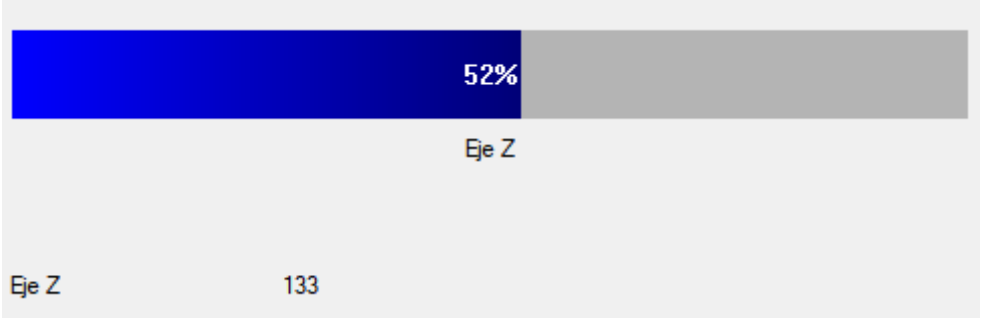

Figure 46 - Z axis at neutral position

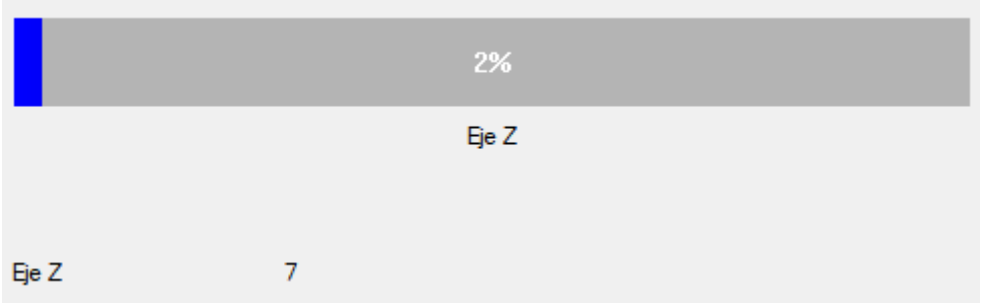

Figure 47 - Z axis at minimum position

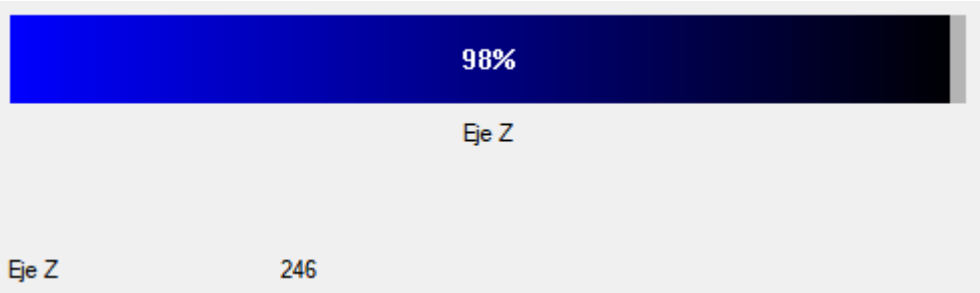

Figure 48 - Z axis at minimum position

While not seen in the images due to the inaccuracy of the joystick, at times the minimum and maximum values were reached for every axis, no overflow was found in the definitive version. During in-development testing an issue was found were at the extreme values the output would fluctuate between minimum and maximum, this was found to be an issue with the native *map* function where if the input was outside the range the output would jump to the other end. This was caused by wrongly setting the input range from 1 to 1024.

With the axes functioning verified we continue with the buttons. This test is performed in the test screen two times, once for each state of the bit precision jumper. Firstly, it was verified that every button could be pressed individually, both those on the breadboard and the joysticks.

Figure 49 - Button 1 and 8 (second joystick) pressed at the same time

Then we verified that every button could be reported as pressed at same time, as it proved impossible to press every button at the same time for a single person this was done by bypassing some of switches.

Figure 50 - Every button pressed

At this moment the only control to be tested is the DPAD, this test is performed while every button is also pressed to further check the ability of sending every control simultaneously.

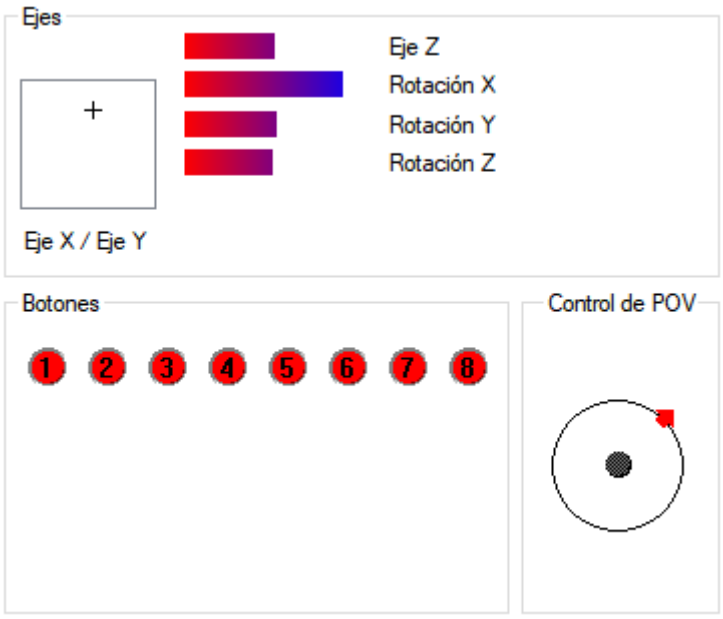

Figure 51 - Up-right on DPAD, every button pressed.

We first check that every of the 8 directions is possible to be pressed in a regular way, meaning no conflicting directions are pressed. Then we for each of the directions we press its contrary while keeping the original one still pressed. All combinations, the expected result and the obtained one are indicated in the following table. Due to the way the loop iterates and determines the direction that is pressed the order in which the switches are pressed does not affect the result as it is calculated from zero on each loop iteration.

Directions pressed	Expected result	Obtained result
No direction	Center	Center
Right	Right	Right
Left	Left	Left
Down	Down	Down
Up	Up	Up
Right, up	Right, up	Right, up
Left, up	Left, up	Left, up
Right, down	Right, down	Right, down
Left, down	Left, down	Left, down
Right, left	Right	Right
Up, down	Up	Up
Right, up, down	Right, up	Up
Left, up, down	Left, up	Left, up
Right, left, up	Right, up	Right, up
Right, left, down	Right, down	Right, down
Right, left, up, down	Right up	Right, up

Table 7 - DPAD testing results

With this test we conclude the artificial testing stage successfully, the device was recognized when connected to the host computer and the controls are interpreted correctly by the system.

Chapter 7.2.2 - Real testing in HID controller compatible software.

The artificial testing of the device can only do so much to verify the functionality, but even with the basic requirements verified we are still to see if the device would perform in a way that makes it viable to use as a control device for real time applications. The objective requirements

to verify with the real testing are the controls to be recognized by a game compatible with generic devices and those controls having a fast enough response not be an impediment.

For the test, a computer game accepting the generic HID controls is selected and some trials defined. The game in question will be BeamNG.Drive by the developer BeamNG. This game aims to be a driving simulator featuring soft body physics and the complete mechanical and electronics behaviour of the vehicles, but the reason it is chosen is the great customizability in its controls and the number of features implemented in the in game vehicles that can be assigned to different inputs that each result in a visual response in game that can be checked, such as a gear being selected or lights being turned on. Some of the functions do support being controlled by axes such like steering. Some of the tests related to button pressing can easily be verified by the visual change occurring in game, while others, mostly related to axes can only be verified by the feeling of the player while using the controls. For each of the tests the requirement is for the action intended to be executed and doing so in a responsive manner.

In order to test the correct recognition of the buttons by the game the following key binds are performed in game:

- Button 1: Turn on/off engine.
- Button 2: Turn on/off lights.
- Button 3: Neutral gear.
- Button 4: Engage/Disengage handbrake.
- Button 5: Next gear
- Button 6: Previous gear.
- Button 7(First joystick switch): Claxon.
- Button 8(Second joystick switch): Look backwards

For testing the DPAD

- Up: move camera closer.
- Down: move camera further.
- Right: next drive mode
- Left: previous drive mode

For testing the joysticks

- X: Steering
- Y: Steering in an alternate test
- RX: Horizontal camera movement
- RY: Vertical camera movement
- Z: Throttle and brake in an alternate test
- RZ: Throttle and brake

With all the actions set up we proceeded to execute them in game confirming the correct response to inputs in a responsive manner. For the case of those controls related to axes where the response cannot be easily perceived as an instantaneous visual stimulus, for this matter the responsiveness was verified by regular gameplay in which the only impediment where the ergonomics of the prototype which are not a main concern in its build.

Taking into account the previously commented results we can determine the development to be successful, as we were able to both obtain a working software able to translate the human interaction into controls recognize by the computer in a responsive manner, and while basic, assemble a prototype that implying that with further knowledge in crafting and more adequate materials it is possible to build a device using an Arduino board for this matter.

Chapter 8 - Addendums

Chapter 8.1 - Conclusions

After finalising the investigation, development, construction and testing we can perform a summary of all the knowledge and results and come with some conclusions.

Firstly, we identified three valid boards from those most commonly available and affordable enough to be considered for a project of this characteristics. The Leonardo was the chosen one as it seemed the most adequate for the kind of prototype that was intended, providing a base it was easy to build upon and a basic set of features whose resulting development could be extrapolated to the other candidate boards. These boards were the Micro, which would be selected by anyone wanting to build a compact device such as a gamepad, and the Due, the most advanced option. The Due would have proven the better choice if the intents of the prototype were to demonstrate the maximum capabilities of an Arduino based device as it would be able to support a greater number of buttons and provide a higher accuracy reading for the axes, 12 bits instead of 10 for a total of 4096 values.

To continue, we explained the development carried in order to accomplish the main objective of obtaining a software for an Arduino board to serve as a controller based in the HID protocol. This development was based in the documentation provided both by the USB Implementers' Forum and Arduino. The inner workings of the code and principles it is based have been explained in such a way the development could be both replicable and adapted to the needs of any user. These adaptations would most likely consist in the use of a different set of controls, let it be in a Leonardo or the other boards featuring more pins, which would result in a different descriptor and report to be constructed according to the instructions. Another modification would be making use of the higher accuracy readings of the Due by modifying the axes translation functions. These modifications would require for both the library and the loop to be rewritten based in the documented knowledge, which would lead to a different release if made public.

Finally, we can comment on the prototype that was constructed. Its intended main purpose was to provide a testing base for the software development, but it ended up shaping it when being designed and constructed in parallel to the software. An important factor in the prototype design was to connect every relevant component, buttons and joysticks, in the easiest and thus most probable by users way, this had to be considered in the software as it affects the way the inputs are to be treated. Another relevant factor in the prototype that was, in this case shared with the software, was that its components, and thus the way they are treated in the software, should be as generic as possible. When selecting the components for the prototype assembly the most weighted criteria for the selection was the ease of acquisition, both the monetary and availability factors. This was made having in mind that if the software was to be used by a third party it would just want the board part to be working no matter the components they use, which would very likely being of different nature if used for the original purpose of the project, flight simulators controls. In that situation the axes would be controlled by a full sized joystick using 2 independent rotational potentiometers, rudder pedals and levers for gas controls and others. What is sent as buttons to the host computer could be any button or position switch, but in both cases, they are most likely to be connected with a single wire and relying in the internal pull-up resistor. This would be done seeking the greater simplification of a system that could be built by soldered wires, and connectors or even relying in PCBs the components are soldered to.

To conclude, we can consider the project a success as all the objectives, both the basic initial ones and the later derived, were accomplished resulting in a valid software for the task and a working prototype that demonstrates the task can be carried with only commonly available means.

Chapter 8.1.1 - Conclusiones

Tras finalizar la investigación, desarrollo, construcción y prueba podemos realizar un resumen de todo el conocimiento y resultados y llegar a unas conclusiones.

Primero, identificamos tres placas validas entre aquellas disponibles de forma más común y suficientemente asequibles para un proyecto de estas características. El Leonardo fue elegido al parecer el más adecuado para el tipo de prototipo a desarrollar, proporcionando una base sobre la que es fácil construir y unas características que darían resultado a un desarrollo fácilmente extrapolable a las otras placas candidatas. Estas placas eran el Micro, que sería el elegido por cualquiera cuya intención fuese construir un dispositivo compacto como un mando, y el Due, la opción más avanzada. El Due podría haber sido la mejor opción si la intención con el prototipo fuese demostrar las máximas capacidades de un dispositivo basado en Arduino. Este soportaría mayor número de botones y proporcionaría mayor precisión en los valores de los ejes, 12 bits en lugar de 10 para un total de 4096 valores.

Para continuar, explicamos el desarrollo realizado para cumplir el objetivo principal de obtener un programa para que una placa Arduino cumpla la función de servir como un controlador basado en el protocolo HID. El desarrollo se basó en la documentación aportada tanto por el USB Implementers' Forum como Arduino. El funcionamiento interno del código y los principios en los que se basa se explican de forma que el desarrollo puede ser tanto replicado como adaptado a las necesidades de cualquier usuario. Estas adaptaciones probablemente consistirían en el uso de un juego distinto de controles, sea para un Leonardo u otra placa con más pines, esto resultaría en la construcción de un descriptor y un reporte distintos que se podrían realizar de acuerdo a las instrucciones. Otra modificación sería hacer uso de la mayor precisión de lectura del Due, se haría modificando las funciones de traducción de ejes. Estas modificaciones implicarían la reescritura tanto de la librería como del bucle de acuerdo al conocimiento documento y conformarían una nueva distribución de realizarse un lanzamiento público.

Finalmente podemos comentar sobre el prototipo construido. Su propósito principal era servir como base para las pruebas del software desarrollado, pero acabo dándole forma a este al ser diseñado y construido en paralelo al programa. Un factor importante en el diseño del prototipo era conectar cada componente relevante, botones y joysticks, de la forma más fácil y por tanto más probable por terceras personas. Esto debía tenerse en cuenta en el software pues afecta a la manera en que las entradas deben tratarse. Otro factor the de relevancia, en este caso compartido con el software, era que los componentes, y por tanto la manera en que se trataban en el software, debían ser lo más genéricos posible. Al seleccionar los componentes el criterio al que más peso se le dio fue la facilidad de adquisición, tanto monetaria como de disponibilidad. Esto se hizo teniendo en cuenta que si el software era usado por una tercera persona esta solo querría del proyecto la parte de la placa funcionando sin importar los componentes que usen, estos probablemente serían de una naturaleza distinta si se usa para el propósito original del proyecto, controles de simulador de vuelo. En esa situación los ejes estarían controlados por un joystick de tamaño completo que haga uso de 2 potenciómetros independientes, pedales para el timón y palancas de gas y otros controles. Aquello enviado al ordenador como botones podría ser tanto un botón como un interruptor posicional, pero en ambos casos, estos se conectarían con un solo cable apoyándose de la resistencia interna de pull-up. Esto se haría buscando la mayor simplificación posible de un sistema que podría construirse por cables soldados y conectores o incluso sobre una placa de circuito impreso sobre la que los componentes estén soldados.

Para concluir, podemos considerar el proyecto un éxito pues todos los objetivos, tanto los iniciales de base, como aquellos derivados posteriormente, fueron cumplidos con la obtención de un software para la tarea como de un prototipo funcional que demuestra que este se puede conseguir por medios comúnmente disponibles.

Chapter 8.2 - Bibliography

[1] J. M. Redondo, "Documentos-modelo para Trabajos de Fin de Grado/Master de la Escuela de Informática de Oviedo," 17 6 2019. [Online]. Available: https://www.researchgate.net/publication/327882831_Plantilla_de_Proyectos_de_Fin_de_Carrera_de_la_Escuela_de_Informatica_de_Oviedo

[2] J. Redondo, "Creación y evaluación de plantillas para trabajos de fin de grado como buena práctica docente.," *Revista de Innovación y Buenas Prácticas Docentes,* p. pp, 2020.

[3] Arduino, "UNO R3 Arduino Documentation," [Online]. Available: https://docs.arduino.cc/hardware/uno-rev3

[4] Arduino, "Arduino Uno Rev 3 - Arduino Official Store," [Online]. Available: https://store.arduino.cc/products/arduino-uno-rev3

[5] Arduino, "analogRead() - Arduino Reference," [Online]. Available: https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/

[6] Arduino, "Due Arduino Documentation," [Online]. Available: https://docs.arduino.cc/hardware/due

[7] Arduino, "Arduino Due - Arduino Official Store," [Online]. Available: https://docs.arduino.cc/hardware/due

[8] Arduino, "Mega 2560 Rev3 Arduino Documentation," [Online]. Available: https://docs.arduino.cc/hardware/mega-2560

[9] Arduino, "Arduino Mega 2560 Rev3 - Arduino Official Store," [Online]. Available: https://store.arduino.cc/products/arduino-mega-2560-rev3

[10] Arduino, "Nano Arduino Documentation," [Online]. Available: https://docs.arduino.cc/hardware/nano

[11] Arduino, "Arduino Nano - Arduino Official Store," [Online]. Available: https://store.arduino.cc/products/arduino-nano

[12] Arduino, "Leonardo Arduino Documentation," [Online]. Available: https://docs.arduino.cc/hardware/leonardo

[13] Arduino, "Arduino Leonardo with Headers - Arduino Official Store," [Online]. Available: https://store.arduino.cc/products/arduino-leonardo-with-headers

[14] Arduino, "Micro Arduino Documentation," [Online]. Available: https://docs.arduino.cc/hardware/micro

[15] Arduino, "Arduino Micro - Arduino Official Store," [Online]. Available: https://store.arduino.cc/products/arduino-micro

[16] Arduino, "Arduinocore-avr," [Online]. Available: https://github.com/arduino/ArduinoCore-avr/tree/master/libraries/HID/src

[17] Arduino, “PluggableUSB and PluggableHID howto,” [Online]. Available: https://github.com/arduino/Arduino/wiki/PluggableUSB-and-PluggableHID-howto

[18] USB Implementers' Forum, “Microsoft Wordd - HID1_11.doc,” [Online]. Available: https://www.usb.org/sites/default/files/hid1_11.pdf

[19] USB Implementers' Forum, “HID Usages and Descriptions,” [Online]. Available: https://usb.org/sites/default/files/hut1_3_0.pdf

[20] darthcloud, “bt_traces/hid_descriptors,” [Online]. Available: https://github.com/darthcloud/bt_traces/tree/master/hid_descriptors

[21] C. Acebal, “Patrón Adapter - Diseño del Software Curso 2021-2022”.

[22] Departamento de Ingeniería Eléctrica, Electrónica, de Computadores y Sistemas Campus de Vieques., TEC Prácticas de Laboratorio 2016-2017.

Chapter 8.3 - Addendums content

Directory	Content
./HID_Controller	Folder containing the complete developed software. In this case it serves both as source code and the distributable intended to be loaded into a board through any IDE supporting Arduino.
./Images	The original photos of the prototype and the software design diagram can be found in this folder.