FC6P01 Project

Final Report

Interactive glove controller

Abstract

In this report it will closely analyse each step of the development process of a Glove Interactive Controller.

My aim was to build a wearable glove controller for computer/user interaction trough hand tilting and fingers movement. The best solution that I could think of is to have a glove with several sensors attached. All these sensors would collect data (such as inclination, movements etc.) and send this data to the computer through any kind (might be wireless -bluet hoot/Wi-Fi- or wired up -I2C-) of connection. On the computer side of things, my aim was to build a C++ wrapper library, which makes it easier for anyone wanting to use the device integrated in any sort of graphic application. The aim of the project is to give the user a new kind of experience by controlling something virtual with their hands.

Contents

1. Introduction

1.1 Subject of the report

This project consists in a wearable glove controller for computer/user interaction through hand tilting and fingers movement.
The glove has several sensors attached to it. All these sensors collect data (such as inclination, movements and finger flexing) and send it to the computer through a USB connection.
The artefact makes use of a micro controller, which has both some built-in sensors and some GPIO (General Purpose Input Output), which allow the external flexing sensor to be attached.

The final product is intended for the use of developers and creators.
To make it easier for people to integrate the device in any sort of graphic application, the controller comes with a library for developers to be downloaded from my GitHub account, which I wrote and packaged personally. The latter allows a quick and easy access to the Glove controller and its sensors.

The project stems from the recent and increasing spread of Virtual / Augmented reality and human-machine interaction. Possibly more and more people will start considering working with these features.
The point of the project is not to take advantage of the market's increasing demand of these types of controllers, but to support the developer community with an affordable and 100% open source device to start with.

From this comes the idea of not simply making the physical product available, but also to provide a quicker setup that makes it less time consuming to start developing.

Accordingly, this report aims to provide detailed explanations on how the project development is being conducted. The following paragraphs will explore the preliminary researches that have been conducted before the project's development and will provide a framework for the layout of the product and the design pattern followed to achieve its final form.

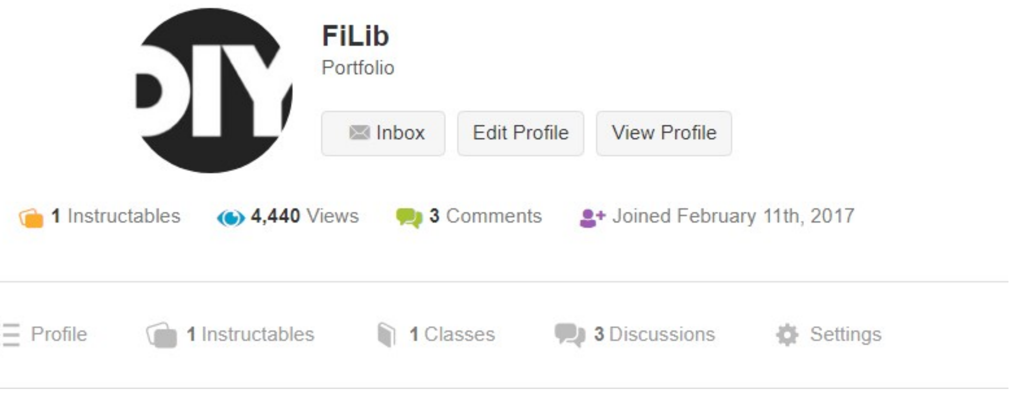


Fig 0: My Instructable page, where I will publish the project

1.2 Structure of the report

The structure of the report will follow and explain each step of the project's development, starting from the choices taken before even beginning the realisation of it up to the final details of how it has all come together.

For what concerns the background research, the related section will mostly focus on the books and sources encompassed during this project. Accordingly, they proved to be essential for both the construction of the device when it came to choose

specific components over others, as well as for the choice of development tools. Once this will have been fully explained, the report continues with the explication of the project design.
The description of the design entails everything, from the layout to the circuit structure and organisation, as well as the design of the actual interface used to access the glove sensors.

After the right selection of design, it will be made clear why and how it all came together to the final product, providing reasonable explanations for why one design would be chosen over another, and giving proof that the decision taken does result to be the most suitable and efficient one.

Finally, the conclusions on the project will be mapped out and will include after-built thoughts of the overall development and considerations of further possible implementations.

2. Background

Thanks to my involvement in the *"BBC Microbit"* foundation last year, my interest towards the world of micro controllers strongly increased, and lead me to study and explore some of the objects that can be realised with such small pieces of electronics.

Microcontrollers are a powerful tool for makers, as they give the chance to control electronics modules by uploading some code onto the board. In a nutshell a Microcontroller is a small computer, so it does have one or more processors cores along with memory and programmable input/output peripherals. When the program is uploaded onto the board it stays there, saved in memory.

There are several microcontrollers on the market, but the features required for the device are basic: neither a large amount of flash memory nor a powerful processor is, in fact, necessary. On the top of this, because there is no need for the device to connect to the Internet I discarded all those micro controllers, which are famous for their Internet capabilities and system stability (Raspberry Pi, Arduino Leonardo, Intel Quark).
This left me with many choices, though the familiarity I had with the Microbit device led me to choose it among other varieties.

The BBC launched this product in the late 2012, but additional partners joined in along the way (including ARM, Microsoft, Samsung).
Every piece of this board is 100% accessible. Accordingly, all companies involved in this project are great supporters of the open source/freeware community.

On the top of this, this micro controller has been given to almost every year-7-student in England and Wales, year-8-student in Northern Ireland, and S1 student in Scotland[1].

Although it is meant to be a teaching device to be approached by beginners, its possibilities of use extend beyond these basic values.

While surfing the web in search of ideas and inspiration on what to do, a website[2] captured my attention. In fact, this provides plenty of available resources, and for each topic a physical book is available for sale.

My interest mostly focused on "*Microbit: IoT in C"* book, this goes through a detailed explanation on how to set up an offline programming environment to upload C++ code onto the device.

Additionally, it tells the user about serial communications, as well as GPIO handling and redirecting. This book has been my main source of inspiration for the realisation of this project.

As for what concerns the programming techniques involved, I learnt most of them along my studies, consulting great website tutorials as well as books like *"Games Programming Patterns"*[3] (where I first learnt about singleton and other OOP patterns). In fact, these projects present more than one technique inspired by the latter, though developed and realised by myself.

[1] http://www.bbc.co.uk/mediacentre/latestnews/2016/bbc-micro-bit-schools-launch
[2] https://iot-programmer.com/
[3] OOP patterns (http://gameprogrammingpatterns.com/)

As well as the above, computer graphics studies helped a lot in topics like memory management (e.g. pointers, references, dynamic memory handling) and data parsing (e.g. buffers, array programming).

Dynamic memory allocation allows the user to manage, resize and reserve memory at runtime, by telling the compiler to keep a certain memory capacity unallocated so that data can be held within it. There are a few articles about C [4] and C++ [5] dynamic memory handling, these websites cover many interesting topics in details and some of the work done in the project is based on these. Especially when it came to integrate two different programming languages together, as this required a lot of transforming ("casting") objects back and forth from a language to another (in the project case C++ and Python).

The methodologies involved will be explained in greater details later on as well as relative approach to them and eventual issues/solutions.

For integrating Python with C++ there is one main official documentation that despite being old is still up to date[6]. I have been studying and documenting mainly from this source as there is no other one available.

Figure 1: The nintendo power glove (image source)

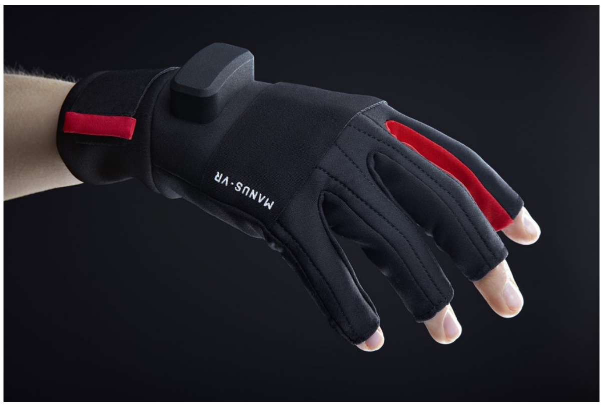
[4] (https://www.includehelp.com/c-programs/dynamic-memory-allocation-examples.aspx)
[5] (https://www.tutorialspoint.com/cplusplus/cpp_dynamic_memory.htm)
[6] (https://docs.python.org/2/extending/extending.html)

Additionally, the founts of inspiration were drawn from a few glove controllers available on the market.

The first one is the Nintendo Power Glove[7]. Considering that it was developed in the late 80's, this device is a pioneer of the augmented reality. Unfortunately, the project failed miserably as the controller ended up being really uncomfortable and difficult to use (Figure 1).

Nowadays there are different kinds of gloves controller but the most famous and probably the most advanced one is the Manus VR[8] . The latter took the glove controllers market to a whole new level, including haptic feedback and water-resistant casing for the glove to be washed if needed.

The Manus VR device (Figure 2) comes with a developer version available on order in two versions: Developer and Professional, costing respectively 1999 and 4999 Euros. It comes with a full Software Developer Kit as well as different plugins for various game engines.



The device is mainly designed to be used in VR applications.

Figure 2: Manus VR (image source)

The following Glove project is though more based on a DIY concept, as I tried to use the smallest number of sensors while succeeding in still obtaining the best results.

[7] (https://en.wikipedia.org/wiki/Power_Glove)

Most of DIY projects[9] available on the web that aim to build a glove, make use of an Arduino for power reason (Arduino board can drive up to 5V) – though I preferred the small size of the Microbit over the compactness and solved the power issue adding an external power source.

I have not found anyone aiming to build a controller for computer applications; most of the tutorial is concerned with developing a glove that drives a robotic arm or hand (using servos and motors).

Everything from the Glove controller interface to the computer has been purely built and ideated by me, and I have not based the architecture or the design on anything but my personal intellectual property.

[8] (https://manus-vr.com/#product-anchor)

[9] (https://celebratelife24x7.wordpress.com/2014/11/17/arduino-project-5robotic-hand/)

3. Design

On a purely aesthetical level, the glove is thought to look as an average black glove and to have all the circuit cables, the batteries and the micro controller hidden and sewed underneath a "cover layer".
The two layers are sewed in together.
Each flex sensor has a dedicated pocket on the corresponding finger, in which they are fit in. The battery, the Microbit, and the recharge module are covered from the external layer, as well as being sewed into the glove textile.

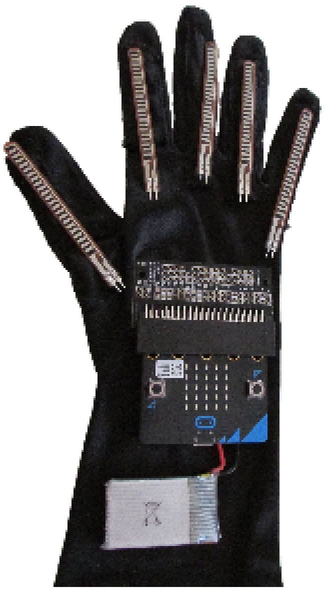


Figure 3: The planned assembly of components

The glove is thought to be as minimalistic as possible.
Two micro USB ports (one for charging and the other one for connecting the device to the computer) are placed on the two diametrically opposed sides of the wrist. The controller makes use of the 5x5 led matrix from the Microbit to display notifications to the user.
The Glove has originally been created as a game controller, but it can be applied to any form of C++ application. It is then up to the developer who is integrating the controller to choose what to do with the retrieved values from the sensors.

In the next paragraphs, I will explain the technical part covering the glove's design system, starting from the Microbit side, the Python interface, up to the final end library (written in C++).
For each I will explain the structure that came out at the end of my research. To

maintain it all more comprised, I divided it into two categories (Software and Hardware) with eventual subcategories when needed.

3.1 Hardware

3.1.1 Microbit

The Microbit makes use of an ARM processor (eg. same as an Iphone) and several built-in sensors such as the followings:
an accelerometer (which returns values accordingly to the oscillation -roll and pitch rot- of the device);
a digital compass;
a low energy Bluetooth module;
a radio antenna (mostly used for Bluetooth from the API's);
5 main pins (3 for digital readings, one for a 3.3V current input (vcc / +) and one for ground(GND / -));
two built in buttons;
a 5x5 LED's matrix;
and both a Power and a micro USB ports.
As shown in figure 3 (above page), a breakout board is necessary in order to attach external sensors to it. The Microbit foundation released different programming environment and different languages the device can be instructed with. The problem with most of these is that the interface simplicity and the visual side are preferred over the actual speed of the code. To reduce the number of layers within the interface and the hardware, it is better not to use a block editor - not Python nor JavaScript. Instead, it is best to use C++ programming language. The reason is that C/C++ is about as fast

as you can get on any machine. It might be possible to get some more speed out of an assembly language version of a program, but it would only be a few percent. Using C/C++ lets programs run tens of times faster.[10]

As the Microbit data access layer is built on the top of ARM mbed, it either needs an offline build system -such as yotta - for offline development or, alternatively, a C++ code should be compiled in .hex files - using the mbed online compiler[11].

3.1.2 Sensors

Flex sensors are analogic resistors. They work as variable analogic voltage dividers. Inside the flex sensor are carbon resistive elements within a thin flexible substrate. When the substrate is bent the sensor produces a resistance output relative to the bend radius. With a typical flex sensor, a flex of 0 degrees will give 10K resistance with a flex of 90 will give 30-40 K ohms.

The 6V battery pack is necessary as the Microbit can only drive 3.3 volt out of the device and the resistance of the sensors is so low that it couldn't even be detected (the tolerated voltage for these kinds of sensors is 5 to 12V).

The batteries are charged through a 1A lithium battery charger module which is a very useful module for DIY projects as it charges through a micro USB cable and it has one led indicator for battery fully charged (blue), as well as one for when it is out of charge or charging (red). This module is extremely cheap and intuitive. It only has 4 pins (two are optional as they would substitute the micro USB voltage in).

Once battery is fully charged there is no need to unplug the module itself as it will directly drive current to the device. If the input voltage is too high, the current will be

[10] https://www.quora.com/Which-is-better-for-microcontroller-programming-Python-or-C++?share=1

less than 1000mA. This is a protection feature: auto-subtract the charging current to avoid burn/damage the chip.

The Microbit reads data from all sensors to then pass it in one big buffer onto the serial. Then there is no further action made from the Microbit as the project is thought in a way that allows only the computer side to access, sort and label the numbers parsed in it.

3.2 Software

Passing data back and forth between two machines is a procedure that can be achieved in many ways (i.e. wiring up a serial connection, using two antennas with any communication protocol -e.g. Bluetooth-, having something connected to the internet and pass everything through a server). I do believe that there is no right or wrong way to do this, but there might be a more or less efficient one.

3.2.1 Generic Architecture Design

To make a good comparison and effectively show the efficiency of the structure planned for the project it is worth comparing it to some already known data flow system. For instance, the Model View Controller (MVC) is an architectural pattern (or design) which aims to separate the application/framework in three logic components: Model, View and Controllers (Figure 4). For each of these, a certain task and

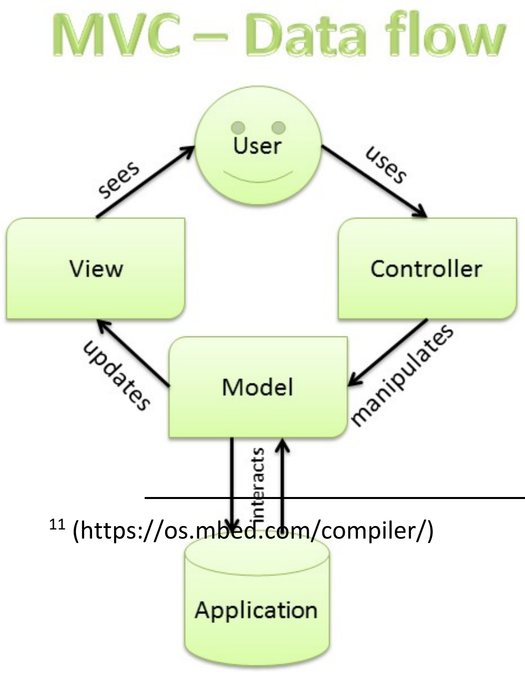


Figure 4: An example of MVC system (image source)

[11] (https://os.mbed.com/compiler/)

specific development aspect of the application itself is assigned.

“MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.” [12]

Without going into details of an MVC system, but just knowing that this is an affirmed data flow system, it is possible to compare it to the one structured for the project in order to highlight and explain differences and similarities.

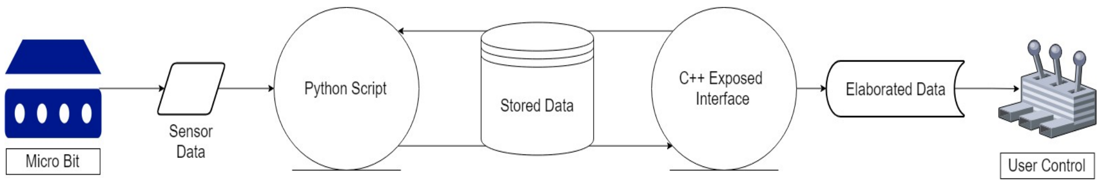


Figure 5: The actual project’s data flow-diagram

First thing that comes evident from the comparison of the diagrammatic representation of the two systems is how the MVC system has two different bidirectional nodes (two points where the data is both received and sent). This is because, in the MVC system, the user is the ending point of the chain, while in my project the final use of the data is unknown, which means it all merges into a “user control” (rather than a user).

To stick to the example, the “View” component of an MVC system is not considered in the project, in fact the data of the Glove Controller have none of what is commonly called “Graphical (User) Interface”, so no representation on the screen.

[12] (https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm)

The Microbit and the Python interface can be considered the Application, which has all the data stored and makes it available to a Model (the C++ interface), the exchanging node is actually between these two.

The bidirectionality results from the Python making the data accessible but not addressing any specific component as it leaves data pending in the memory. Here comes in the C++ side, which aims for that place in memory, accesses the data and requires the Python script to start updating.

Finally, it will retrieve all the required data and expose it (as to make it accessible) through some getter functions (simple and specific requests) to anyone who is willing to implement the glove to his own application.

If I were to take both the final graphic application within which the glove controller is used and the user into consideration, it would then end up looking like the one in figure. 6.

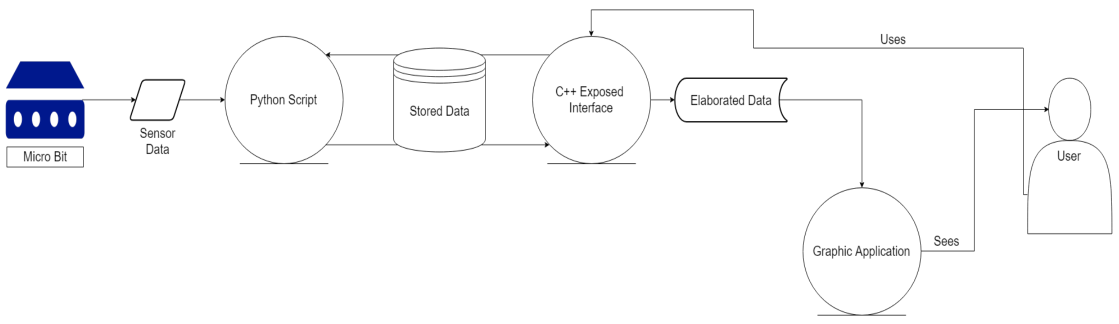


Figure 6: Extended system flow

3.2.2 Python Script

The organization of the Python script is really important because it will exclusively work as a bridge. Accordingly it will need to run independently both from the Microbit and from the C++ interface.

To start working on the application as quickly as possible, it seemed logic to use a Serial library[13] to do the basic serial operations and encapsulate it in a class called Serializer.

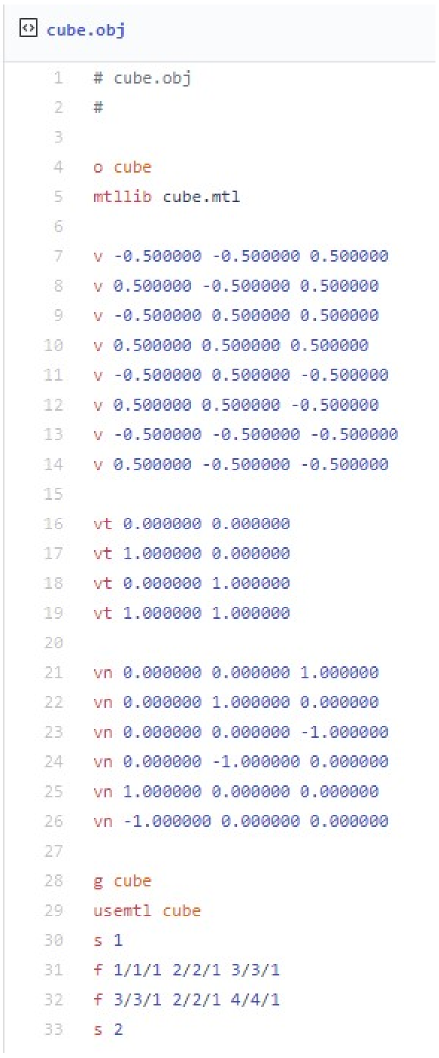
The choice of which library should be used is an important one, as most of them are products of hard work so usually very well optimized.

The Serializer makes use of the singleton design pattern. It is nothing more than a class with the ability of replicating itself in the constructor, so that every time it needs to be called, it parses itself as a referenced copy without taking any extra memory.

There are many debates around this topic in that it is thought to be a bad way to deal with data, and it goes against Object Oriented Programming logic by making every variable public and accessible from the outside.

The serializer is declared as a normal Python class but "decorated" as a singleton. The two designs together result in a handy and reusable serializer that only needs instantiation once. The actual techniques will be shown in the Implementation section, further on in this report.

Any exchanged data between the components of the entire architecture is parsed with a specified token next to it, in the same way an obj file is written (figure. 7).


cube.obj

```
# cube.obj
#

o cube
mtllib cube.mtl

v -0.500000 -0.500000 0.500000
v 0.500000 -0.500000 0.500000
v -0.500000 0.500000 0.500000
v 0.500000 0.500000 0.500000
v -0.500000 0.500000 -0.500000
v 0.500000 0.500000 -0.500000
v -0.500000 -0.500000 -0.500000
v 0.500000 -0.500000 -0.500000

vt 0.000000 0.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 1.000000

vn 0.000000 0.000000 1.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 -1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000

g cube
usemtl cube
s 1
f 1/1/1 2/2/1 3/3/1
f 3/3/1 2/2/1 4/4/1
s 2
```

Figure 7: Example of a cube obj, (image source)

Any .obj model file, if opened, is just a sequence of numbers preceded from a token (a

|/

letter) that specifies what the following data is going to represent (v = vertices, vn = normal etc.).
In the same way the data parsed between the interfaces of the project is recognised using this methodology. This process is quick and efficient, and it does not require the program to compute extremely complexed calculations in order to evaluate what data type to expect for each line parsed in, as this can explicitly be declared via that token.

3.2.3 C++ End Interface

Having the final side written in C++ is quite a convenient thing as most of the games nowadays are developed in either C++ or C#.
Anyway, following the logic planned so far, if for any reason one needs to retrieve those values in R or JavaScript, the developer will still be able to just remove the last layer and use the Python script instead. Of course, given user will have to manually retrieve them and put them in the right order, but that's a whole other story.
Python comes with a C++ library, through which is possible to call the interpreter, run a script and even call functions inside the script.
The latter library is quite old style and the access to a Python object like a method (or even just a variable) is required each time. The Python object, in turn, needs to be casted into a C++ object. This (as previously mentioned) is a very sensitive procedure as Python is a very dynamic language, whereas C++ is not.

Because the C++ needs to access the Python scripts and it needs to ask for the variable for each single frame, it will also need to wait for the script to answer back. This is a common problem when it comes to synchronizing two entities communicating with one another (e.g. networking: side/client).

Multithreading is only one of the solutions to this problem but (because it is mostly supported by all machines and because there is an official C++ library to use threads) it seemed to be the most suitable for the glove controller system.

Multithreading is the ability of a CPU (Central Processing Unit) to execute multiple processes concurrently.
"The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. […] Thus, techniques that improve the throughput of all tasks result in overall performance gains"[14].
In the end user interface, threads have initially thought to be contained in a resizable container. This last implementation of the design resulted in some problems, which led me to go back to it and create a much simpler and clearer design of the thread handling.
The first design was thought so that every time a thread is needed to do something, it can be created, initialized, and assigned to a function and detached from the main program flow.
This resulted in errors when the program was to delete the thread pointers from the container. More specifically:
Once it has been created, no pointer will have a value until it is initialized with the *new* keyword: to each *new* keyword a *delete* needs to match. Because C++ does not have a garbage collector, all the memory that has not been manually erased -once unnecessary- will remain there.
This usually means that new instantiated objects won't be able to take that single

[14] (https://en.wikipedia.org/wiki/Multithreading_(computer_architecture))

place, and will instead be stored into the next available one.
After a certain number of program iterations (depending on the machine hard memory specifications) the whole executable will eventually crash.
When I created, initialized and pushed a thread’s pointer into the container, this was fine. Though when it came to delete that pointer it was too hard to keep track of both emptying the array slot containing the thread as well as the thread’s pointer.
The main idea quickly changed shape into something simpler: just two threads are created from the start and any extra operation is kept within the thread from which has been required.

In addition to the threads that will handle the communication to the Python, there is the procedure of opening the Python interpreter, initialise it and shut it down once it is done, this should not require a lot of coding if achieved properly.

4. Implementation

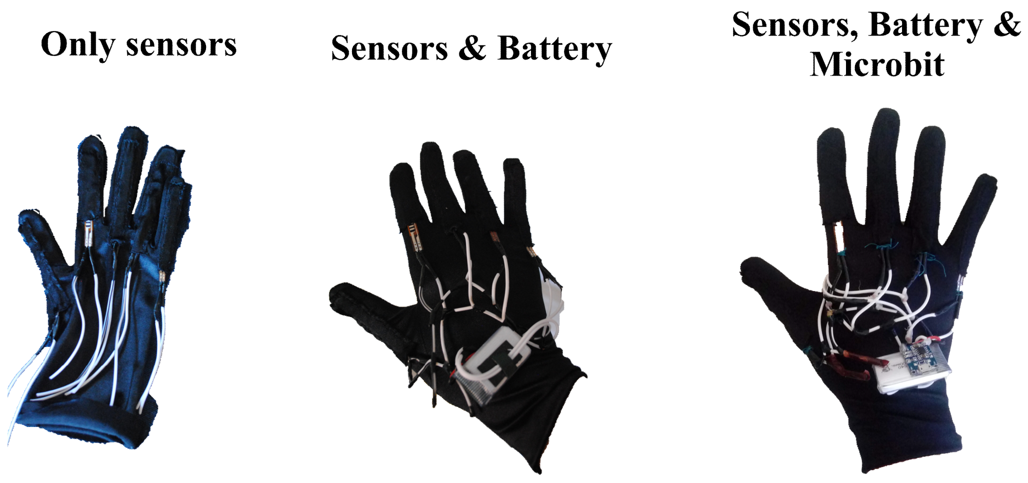


Figure 8: three different states of assembling

4.1 Software

4.1.1 Microbit

As mentioned above, the Microbit has been instructed in C++ for speed's purpose. The whole set up of the environment required some time to be implemented.

The C++ Microbit runtime has been developed by Lancaster University and contains device drivers for the device and gives access to a suit of mechanisms to make programming the Microbit easier.

Because the whole is built on top of ARM mbed, it makes use of yotta to link all the building dependencies. This means fetching the right components -compiler (Gcc in the project case), libraries and eventual toolchains- and instruct them to compile the C++ code to a hexadecimal file (readable by the device).

Given that yotta needs to link to the Python interpreter too, and given that I use Python for external development and different applications, it has been necessary to build a virtual environment and install only the libraries needed by the built system.

To make the development faster, I wrote some bash script to activate the virtual environment, build the C++ and upload it, all in one go. The procedure it looks like this anyway:

C:\yotta-venv\Scripts\activate => Activate the virtual environment

cd Path\To\Program\ => Access the directory of the code to be compiled

yotta build => Build it

cp \build\bbc-microbit-classic-gcc\source\program name-combined.hex F:\ => Copy it to the F drive (the Microbit).

Once this set up was done, making it work, change the code, and upload it onto the Microbit became way easier.

The program running on the device is simple and only sees three main functions doing all the job: initialise, loop and release.

Because the only library that can be included in the project is the Microbit official one (trough which is possible to instruct the device and access its pins and sensor), I needed to write a map function myself. The latter takes two variable and maps them to another specified range (for the project example it takes the returned analogic read from the sensors and maps it between 0 and 90 degrees of bending):

```
long map(long x, long in_min, long in_max, long out_min, long out_max) {
       return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
       }
```

The ‘initialise’ and ‘release’ functions respectively initialise and shut down the serial communication. The loop on the other hand is where the actual variables get sent using the tokenisation method mentioned before.

This sends a “BEGIN” before sending the set of variables (tilting x, tilting y, 5 fingers flex value and two buttons pressed Boolean value). Before and after each variable a “b” and an “e” are sent so the final interface knows that it will need to loop in between the two finds of the “BEGIN” and it will look variables only between a “b” and an “e”.

The final implementation:

```
uBit.serial.send("BEGIN");

uBit.serial.send(" \n");
uBit.serial.send("b");
uBit.serial.send(uBit.accelerometer.getPitch());
uBit.serial.send("e");

...

uBit.serial.send(" \n");
uBit.serial.send("b");
uBit.serial.send(angles[1]);
uBit.serial.send("e");
```

4.1.2 Python Script

For what concerns the Python serial interface, on the other hand, a few more functions have been implemented.

In this script is worth looking at how a singleton is implemented and how in this case it decorates the serializer too.

```
def singleton(cls, *args, **kw):
    instances = {} #create an empty array
    def _singleton(): # define a temporary singleton instance

#if the parsed class (cls) has not already being assign a singleton
```

```
        if cls not in instances:
            instances[cls] = cls(*args, **kw) #create for it a new copy of itself
        return instances[cls] #and return it
    return _singleton
#above if the parsed class has already been copied reurn it as it is

#the following line is a decorator in Python
#this will make the following class a singleton
@singleton
class MySerializer:
    def __init__(self, _baudrate=9600, _timeout=100):
        self._ser = serial.Serial()
        self._ser.baudrate = _baudrate
        self._ser.timeout = _timeout
```

The above text shows a comment for each line that should explain what each of them does.

The class serializer is only shown with the constructor (__init__) but it does have many other tools functions for opening, closing, checking if open and so on.

The actual core of the scripts when ran first requires to know what ports are available. Consequently, I wrote a check system ports function which scans all the COM's and only returns the available ones.

```
def checkSystemPorts():

    if sys.platform.startswith('win'):
        ports = ['COM%s' % (i + 1) for i in range(256)]
    elif sys.platform.startswith('linux') or sys.platform.startswith('cygwin'):
        # this excludes your current terminal "/dev/tty"
        ports = glob.glob('/dev/tty[A-Za-z]*')
    elif sys.platform.startswith('darwin'):
        ports = glob.glob('/dev/tty.*')
    else:
        raise EnvironmentError('Unsupported platform')
    return ports
```

The above few lines check as well for what operative system the program is running on. This is not used as a feature now as the system is designed for Windows. Although it is not possible at the moment to run the project on different OS's, all the libraries used (as well as the programming laguages) are platform independent and the whole system has been arranged so that making it completely cross platform on different Operative Systems would not take too long.

Because the mentioned function returns an array of ports, it is possible to iterate trough this last and try to connect to each one of them. When one port allows connection, then the script is going to establish connection, this is achieved in the 'serial connect' function.

```
def Serial_connect(_br):
    ser = MySerializer()
    for port in checkSystemPorts():
        try:
            ser.SetPort(port)
            ser.open()
            return "True"
        except (OSError, serial.SerialException):
            pass

    if not ser.connectionAvailable():
        return "False"
```

Python has some very helpful features when it comes to error trap, as all the exceptions thrown can be caught, and accordingly modified afterwards.

For instance, in the above lines, the serial connection happens only if the port is available. If this is not the case, an exception would be raised, caught by the program and excepted by doing nothing ("pass"). In case the port is active and can be connected to, the function will return true. If none of the ports has been found available it will return false.

Once everything has been initialised and the connection has been safely established, it is possible to ask the script for the variables, the function to do this is the following.

```
def GetVars():
    ser = MySerializer()
    checking = 1
    lines = []

    while checking:
        lines.append(ser.recv())
        if lines[0] == "BEGIN \n":
            checking = 0
        lines = []

    for value in range(0, VARIABLES_TO_BE_READ):
        lines.append(ser.recv())

        if lines[0] == "SHUTDOWN\n":
            ProgramOn = 0

        if lines[0] == "STOP\n":
            ProgramOn = 0


    return str(lines).strip('[]')
```

The function initialises an empty array, sets the checking Boolean variable to true and then starts a loop until it reaches the next "BEGIN" token. This is done to discard all the values up to the start of the list ("Begin") and to ensure the synchronicity of the two parts.

Once the token has been found, for a certain number of times (the same as the number of variables) it will read a single line, this bunch of lines will contain all the variables needed. After having stored them all in the previously initialised array it returns the whole as a list of string.

The above-mentioned functionalities pretty much make the whole script.

4.2.3 C++ End Interface

Finally, the C++ interface, has a way more complicated implementation as it needs to manage all the threads as well as handle the communication with the Python script.

The interface only exposes its public members function, while keeping all the other belonging objects private and inaccessible from the outside.

First, it is worth looking at the private variables needed to hold the objects the interface needs:

```
class MyExtension

{

private:
	/*Default path to the Python script*/
	std::string _path;
	/* Create some Python objects that will later be assigned values*/
	PyObject *pName, *pModule, *pDict, *pFunc, *pArgs, *pValue;
```

The series of Python object (this type of object is built in the Python.h library for C++) will respectively contain: the path to the Python file, external modules to be imported in the script, dictionary (container) holding eventual variables, function names with its arguments and a simple Python value which can be used to temporarily store any data. Many other private members hold: all the sensors value, Boolean for connection established, a tool function to convert string to numbers and some thread functions such as connect, close and so on.

```
/*The two main threads*/
std::thread * Connecting_thread;
std::thread * Updating_thread;

/*Threads function*/
void Connect();
void UpdateVariablesFromSerial();

/*Function for shutting everything down*/
void SerialClose();

/*Private Tool Function*/
float StringToNumber(std::string s) {
	int Numb;
	std::stringstream str(s);
	str >> Numb;
	return float(Numb);
}
```

All the sensible private variables have corresponding getters.

In Object orientated programming, when a class needs to expose some members to the outside, it is a good practice to keep that variable private to the class and create public functions to get or/and set the variable. In a nutshell, the reason of this procedure is that in a more complex system (extended with multiple classes and inheritances between them) the compiler gets confused if everything is public and accessible from anywhere. If the programmer explicitly declares certain variables or methods to be only accessible from within a single class, all those won't be even taken in consideration when compiled and linked to the external classes.

The following are the remaining sensible variables. Each of them has at least one line of comment to understand what they are going to hold.

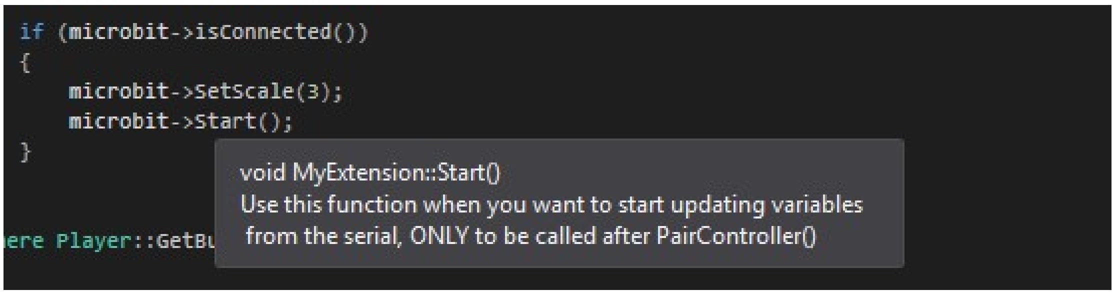
```
/*Private sensor values*/

float pitch, roll;

/*Private Array of current fingers Flex value*/
float fingers_value[MAX_FINGERS];

/*Private bools for buttons value*/
bool button_A_pressed, button_B_pressed;

/*Private scale and timeout*/
float scale;
float timeout;

/*This static variable is shared between instances of microbit Extension.*/
static bool connected;

/*Time since pairing mode started */
double duration;
```

The bool "connected" needs the static attribute in the eventuality where more than one instance of the extension is created.

Static is a C++ keyword, which specifies that the following declared variable is going to be mutually shared between the instances of a same class. For example, let's say that the maker using this library with the attached device needs to divide the pairing state from the game play.

Though, as the maker will need to create an instance of the extension in both the states, each instantiation will need to know if the connection to the Python script has already been initialised. This is represented by the connected variable, which is turned true when both the connections between the Python and the C++, and between the Python and the Microbit, will be established.

All the other methods from the extension are public and accessible from the outside. As shown in Figure 9, when a function from the library is written down, if hovered with the mouse cursor it will display the respective comment.



```
if (microbit->isConnected())
{
    microbit->SetScale(3);
    microbit->Start();
}

ere Player::GetBu
```

```
void MyExtension::Start()
Use this function when you want to start updating variables
 from the serial, ONLY to be called after PairController()
```

Figure 9: The start function description is shown when the mouse is over it

The followings are all the public methods and some getters:

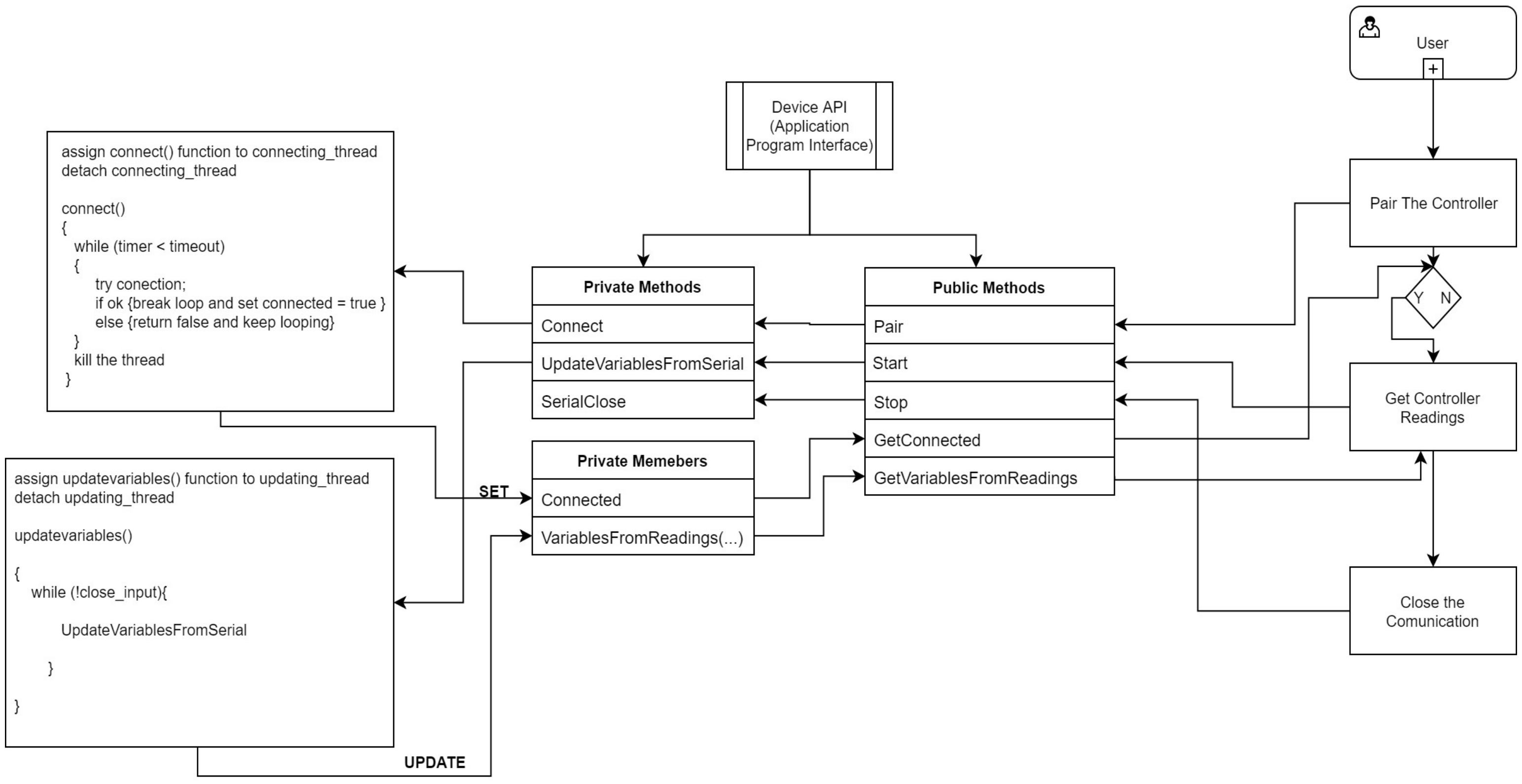
```
public:

       /*Use this function when you want to start updating variables
       from the serial, ONLY to be called after PairController() */
       void Start();

       /*Use this function when you want to stop updating variables
       from the serial, this will automatically both close the serial
       and destroy the Python interpreter*/
       void Stop();

       /*Contructor with default parameter assigned*/
       MyExtension(const std::string& path);

       /*Deconstructor*/
       ~MyExtension();

       /*Function to be called when entering pairing mode, it will look for
       connection until when the timeout in SECONDS is reached*/
       void PairController(float _timeout);
```

```
        /*Enum of fingers for indexing*/
        enum fingers { F_THUMB, F_INDEX, F_MIDDLE, F_RING, F_PINKY };


/******************* INLINE SETTERS AND GETTERS****************************/

        /*Set Baud rate*/
        inline void SetBaud(long _baud) {
               baud = _baud;
        };


        /*Returns the flex of the specified finger*/
        inline float GetFlex(fingers f) { return fingers_value[f]; };

        /*True if connection established, false otherwise*/
        inline bool isConnected() { return connected; };

        …etc…
```

As shown above, when it came to get and handle fingers value, both an array of "fingers_value "(private) and an enumeration of type fingers have been created. C++ Enumerations are a very effective tool, which allows you to create a new variable and give it a type name ("fingers" in this case). Each enumerator is constructed with an enum-list, which is just a list for number to which a string key is associated. This allows anyone who is willing to access the fingers_value array of floating point numbers to index it with a clear and understandable key instead of just guessing a number from 1 to 5. E.G:

```
microbit->GetFlex(microbit->F_THUMB);
```

The main interface methods (`Connect`, `UpdateVariablesFromSerial` and `SerialClose`) are the core of the whole extension, and are hidden behind the public Pair, Start and Stop methods. The logic is the following:



```
assign connect() function to connecting_thread
detach connecting_thread

connect()
{
  while (timer < timeout)
  {
      try conection;
      if ok {break loop and set connected = true }
      else {return false and keep looping}
  }
  kill the thread
 }
```

```
assign updatevariables() function to updating_thread
detach updating_thread

updatevariables()

{
   while (!close_input){

        UpdateVariablesFromSerial

      }

}
```

Figure 10: Diagrammatic representation of end user interface

When a call to Pair is made, it will be assigned a timeout in seconds as a parameter. The Pair function will initialise a thread and assign it the Connect function. From the moment of the call to the next timeout seconds the thread will run “detached” from the main program flow. It will keep looking for the connection to happen by waiting for the Python script function “Serial_open” to return true.

As soon as this happens the thread will shut itself down and return true, if the timeout is reached and the thread is still alive, it will break the loop and exit the function normally (what shown next is not the whole function but only the main bits):

```
clock_t start;
	duration = 0.0;
	start = clock();

	while (duration < timeout)
	{

		// Call the serial connect function.
		pFunc = PyDict_GetItemString(pDict, "Serial_connect");
					...
		try
		{
			std::string result;
			result = PyString_AsString(pResult);

			result == "True" ? connected = true : connected = false;

					...

			duration = (clock() - start) / (double)CLOCKS_PER_SEC;
					//std::cout << duration;

		}

		catch (const PyBaseExceptionObject& e)
		{
			std::cout << e.message;
			throw e;
			break;
		}
	}
	delete Connecting_thread;
	return;
```

After the Pair function has been called and the connection has happened, the user will be able to call the Start function, which will do the same operation as the connect and launch a new thread (updating_thread).

The Start function will assign the thread the UpdateVariablesFromSerial function. The latter will run detached until the call of Stop and will as the Python script to update the variables every frame.

Between the Start and the Stop function, for the user, will be safe to ask for any variable trough the getters methods.

The reading of the variable always uses the tokenization, but this time only between the “b” and the “e” as the Python will have already discarded everything else by now.

```
	/*Variable that will hold the whole buffer of chars*/
	std::string result;

	/*Get the function resul and cast it from Python Object to std::string*/
	result = PyString_AsString(pResult);

	/*Temporary variables to hold the string value before converting it*/
	std::string str_pitch, str_roll, str_thumb, str_index, str_middle, 
str_ring, str_pinky, str_btnA, str_btnB;

	/*Start reading whats in between the b(egin) - e(nd) tokens*/
	str_thumb = result.substr(result.find("b") + 1, result.find("e") - 1);
	fingers_value[F_THUMB] = StringToNumber(str_thumb);
	result.erase(0, result.find("e") + 1); //Clear up to where has already 
been read
```

4.2 Hardware

Once all the plan of what to use in terms of hardware and once I had all the software side working efficiently, I could solder all the components together.

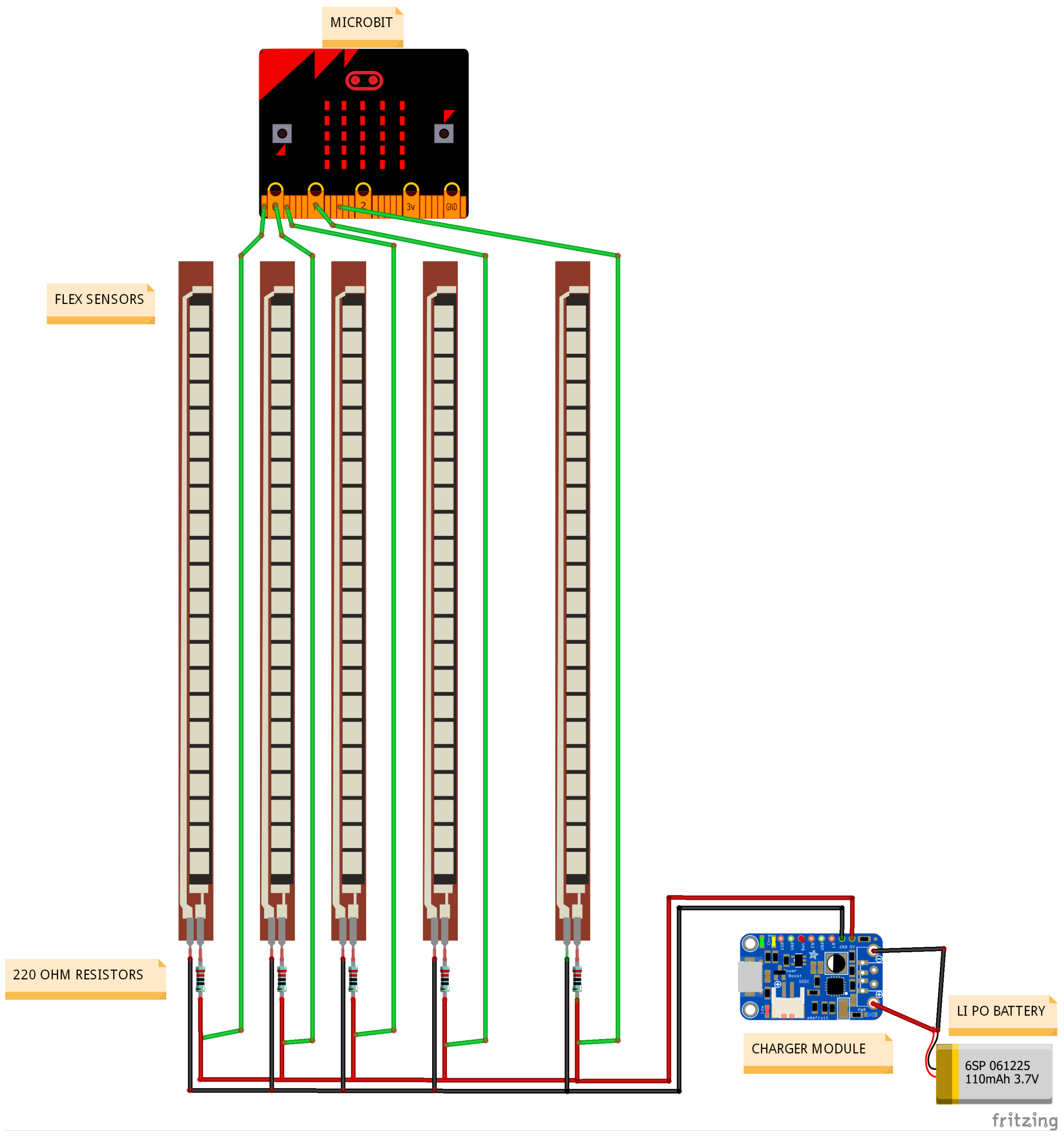


Figure 11: Fritzing of the final design with labels

Once a first plan of design has been created (figure. 11), it was just a matter as to where to put which component and how to solder it properly.

Logically all the components need to fit in one hand and sewed onto the textile. This required some external help, but eventually it led to effective results.

I first soldered all the 220 Ohm resistors to each sensor leg (Figure 12).

Once this was done for all the sensors, I put them together with the glove (Figure 13) and then wired each cable to the Microbit and the battery.

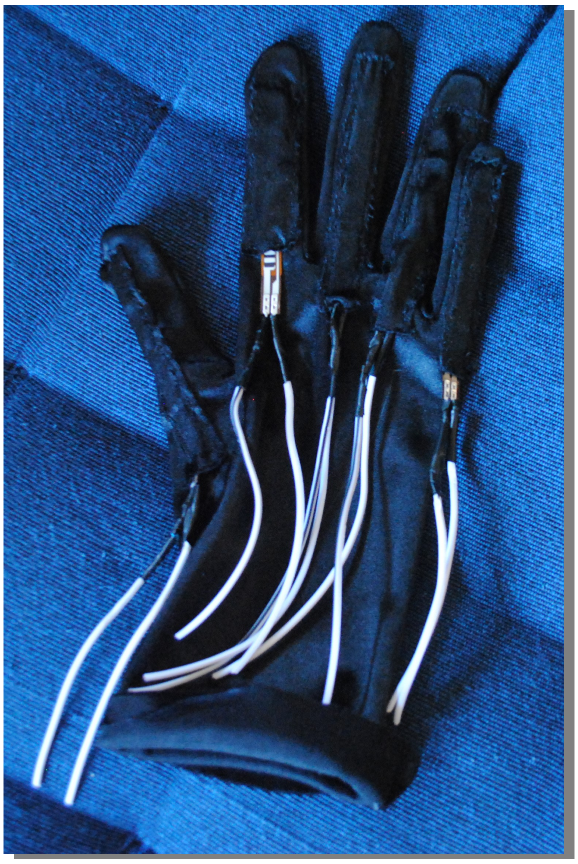
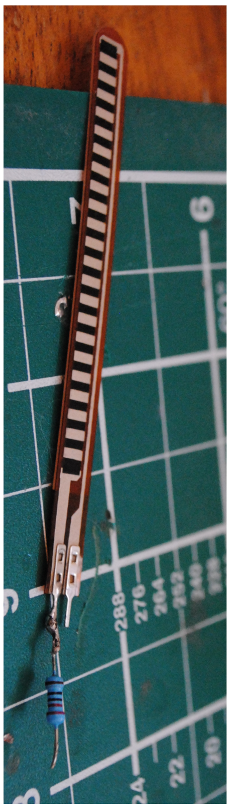


Figure 12: Glove with flex in place

Once all sensors were in the right place



on the glove, I wired the battery to the charger module. This is required to solder the latter first and then add the wires going to the sensor (Figure 12).

Figure 14: Resistor soldered to the flex sensor

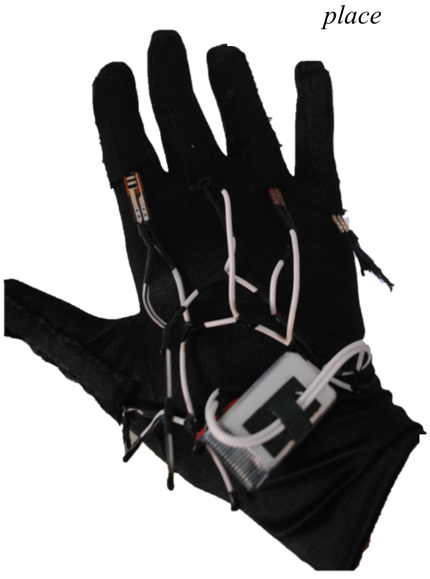


Figure 13: The Glove with both battery and sensors attached

After the connection between the battery and the sensors, and between the battery and the charger was ready, the Microbit was added. Each selected pin on the Microbit was dedicated to analogic values from the Microbit pins map[15].

When it came to get a breakout board to access all the Microbit pins, I purchased a solderless one so that only the pins needed by the project would be soldered by myself, while the others would not be active at all (figure 15).

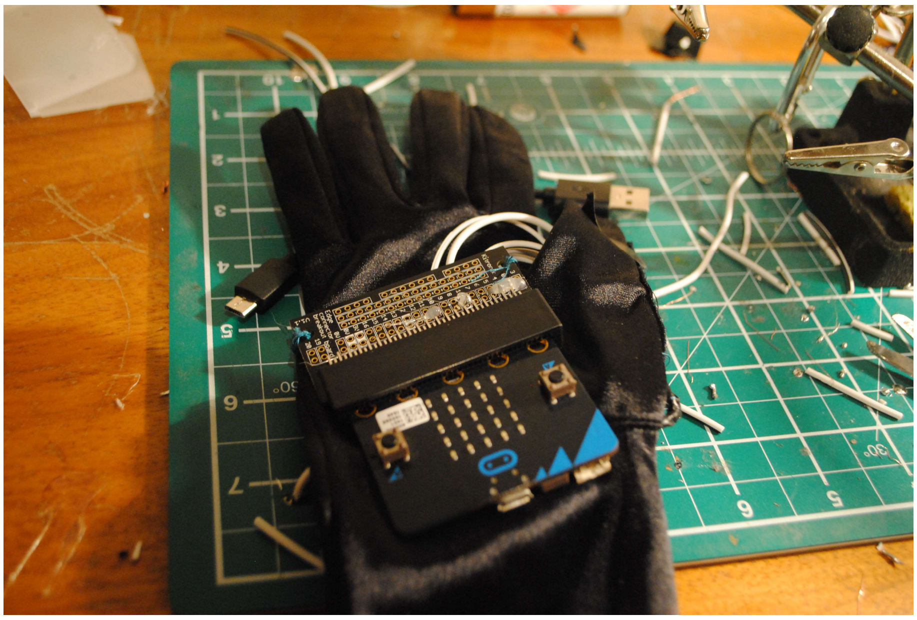


Figure 15: Only 5 pins are accessed on the breakout board

Once everything has been soldered in place, the last thing to be done is to cover everything with an external layer of textile and sew the two layers together.

[15] https://blog.gbaman.info/wp-content/uploads/2016/02/600xNxpin_mapping_0.6.png.pagespeed.ic_.NlEPaMNpEG.png

5. Testing & Conclusions

The statements provded when I first started the project and explained in the introduction of this report have been fully matched. The controller works and the whole interface as well, they will both be published soon on my GitHub and my Instructable page. This will allow whoever wants to, to replicate and reproduce the device, and then makes use of the library to actually build projects with it integrated. The components chosen resulted to perfectly suit the project's needs. The proper functionality of the extension for the device shows the planned architecture of the system successfully.

The project could be enriched of other readings from the built-in sensors of the MicroBit device but having the possibility to move with hand's tilting and to do great things with the flexing sensors of the fingers is good enough as a starting point.

The aim of this project was to make a controller for machine-user interaction and to wrap up the way I did it so that I (or whoever wants to) can use it in the future.

Something that has been considered in the very first few drafts of the projects is the finger touch. This was supposedly going to be achieved through a RFID reader placed on the palm of the hand (and the MicroBit on the top). Each finger would then have had a RFID tag on the fingertip. The problems resulted from the first testing sessions where quite a few: the RFID reader (due to its size) would not fit in a palm of your hand and neither would the smallest (and reasonably affordable) RFID tag.

The battery as well wasn't originally planned. The reasons I decided to use an external power supply are mainly two:

-The flex sensors require it to give constant values. I first tried to only provide them

with the 3.3v from the MicroBit, but some of them wouldn't return the right values.
-In a future implementation the data could be parsed wirelessly and the MicroBit would then need to drain current off the battery instead of the USB.
The flex sensors also gave some trouble just because they are not as precise as I would expect them to be. Plus, they are very sensible to any movement and, every now and then, they do return some garbage value out of nothing. Also, I firstly bought one sensor for testing only, and only afterwards I bought the other 4. For some reason the one that I worked on before returned different values than the others.
Anyway, they are all mapped to the same final values, no matter what the original ones are.

Overall the project resulted to be challenging and maybe if I were to go back I would have done something graphically appealing rather than a more mechanical product. It is very satisfying to have the controller working and it is my intention to also provide it a proper graphics application with 3d modelled hand and the values of the sensors correctly mapped for a realistic finger movement in virtual world.
Future implementations are possible, some of them regard the number of readings returned from the device and some other might even see the device reassembled with different circuits and possibly another kind of sensor for the flex detection.
I am satisfied with the final product, but in a further implementation, it will probably have different kinds of components. I would probably choose a wearable controller which allows to be sewed on textile more easily and probably a different brand for the flex sensors. For what concerns the design of the library system, that won't ever change, as I think I hit a good point there and I really like the way it works overall.
Developing it has been a lot of fun and I came across different new topics, which I

hadn’t researched before. The main satisfaction about the software side comes from the complete originality of it, as in order to build it I did not follow any tutorial or external guidelines.

7. References

-This website explains how to properly configure a development environment for the Micro:bit runtime-

Microbit Offline Toolchains - https://lancaster-university.github.io/Microbit-docs/offline-toolchains/

-Image of the Microbit Pins map- https://blog.gbaman.info/wp-content/uploads/2016/02/600xNxpin_mapping_0.6.png.pagespeed.ic_.NlEPaMNpEG.png

-To study a bit more about mbed devices and how to compile code for ARM processor-based boards-

Mbed OS Documentation - https://os.mbed.com/docs/v5.6/introduction/index.html

-To endorse my C++ skills and for checking any issue due to synthax-

C++ tutorials point - https://www.tutorialspoint.com/cplusplus/

-To refresh programming design patterns such as singleton-

Design patterns - http://gameprogrammingpatterns.com/

-For solving issues related to Python programming this is the official documentation website-

Python docs - https://www.Python.org/doc/

-Manus VR official Website- https://manus-vr.com/#product-anchor

Fairhead, H., 2016. Micro:bit IoT In C. 1st ed. England: I/O Press.

Platt, C., 2015. Make: Electronics: Learning Through Discovery. 2nd ed. England: Maker Media.

Josuttis, N., 2012. C++ Standard Library, The: A Tutorial and Reference. 2nd ed. England: Addison-Wesley Professional.