

EPFL

Laboratory of Intelligent Systems

Semester project

A Modular Hardware Front-End for fast Prototyping

*Responsibles:*

*Author:* Prof. Dario Floreano

Cyrill Lippuner Matteo Macchini

Olexandr Gudozhnik

Contents

[1 Introduction 5](#_Toc534816393)

[2 State of the art 6](#_Toc534816394)

[2.1 ROS 6](#_Toc534816395)

[2.2 Arduino 6](#_Toc534816396)

[2.3 LabView 6](#_Toc534816397)

[2.4 Universal Windows USB Drivers 7](#_Toc534816398)

[3 What is already there? 8](#_Toc534816399)

[3.1 Hardware & Protocols 8](#_Toc534816400)

[3.2 Software 8](#_Toc534816401)

[4 What is new? 9](#_Toc534816402)

[5 Project Structure 10](#_Toc534816403)

[5.1.1 GitHub 10](#_Toc534816404)

[5.1.2 Docs 10](#_Toc534816405)

[5.1.3 Setup & Installation 11](#_Toc534816406)

[6 Interface Architecture 12](#_Toc534816407)

[6.1 Main 12](#_Toc534816408)

[6.2 Boards & Communication Methods 13](#_Toc534816409)

[6.3 GUI 13](#_Toc534816410)

[6.3.1 Board Overview 13](#_Toc534816411)

[6.3.2 Device List 13](#_Toc534816412)

[6.3.3 Device Settings 13](#_Toc534816413)

[6.3.4 Live Data Visualization 14](#_Toc534816414)

[6.4 Communication to third-party software 14](#_Toc534816415)

[6.5 Diagnosis 15](#_Toc534816416)

[7 Firmware Architecture 16](#_Toc534816417)

[7.1 Main 16](#_Toc534816418)

[7.1.1 Scan Thread 17](#_Toc534816419)

[7.1.2 Update Thread 17](#_Toc534816420)

[7.1.3 Communication Thread 17](#_Toc534816421)

[7.2 Configuration 18](#_Toc534816422)

[7.2.1 Software Configuration 18](#_Toc534816423)

[7.2.2 Hardware Configuration 18](#_Toc534816424)

[7.3 Drivers 19](#_Toc534816425)

[7.3.1 Device Drivers 20](#_Toc534816426)

[7.3.2 Mux Drivers 20](#_Toc534816427)

[7.4 Communication 21](#_Toc534816428)

[7.4.1 API 21](#_Toc534816429)

[7.4.2 Embedded Scripting 21](#_Toc534816430)

[7.5 Diagnosis 22](#_Toc534816431)

[8 Demos 23](#_Toc534816432)

[8.1 Data Acquisition, Visualization and Streaming 23](#_Toc534816433)

[8.2 Plug’n’Play and Drivers 24](#_Toc534816434)

[8.3 API and Embedded Scripting 24](#_Toc534816435)

[8.3.1 Semester Projects using the API 24](#_Toc534816436)

[8.3.2 Simple Commands 26](#_Toc534816437)

[8.3.3 Connecting Two Sensors 27](#_Toc534816438)

[8.4 Diagnostics 29](#_Toc534816439)

[9 Evaluation 31](#_Toc534816440)

[10 Future Work 33](#_Toc534816441)

[10.1 Driver Installer and Custom Scanning 33](#_Toc534816442)

[10.2 Embedded Scripting 33](#_Toc534816443)

[10.3 SPI & UART 34](#_Toc534816444)

[10.4 C / C++ 35](#_Toc534816445)

[11 Conclusion 36](#_Toc534816446)

[12 References 37](#_Toc534816447)

# Introduction

Although a relatively new technology, wearable sensors have become the object of interest for several industrial, research and clinical applications [1] [2] [3]. Despite the large research potential of these sensors, there is no complete and versatile environment for interfacing such devices.

The vast majority of sensors available support standard serial data-link protocols, but they lack any common network or transport layers. Because of this, any potential user should either create his own driver, use an existing open-source solution or, in some cases, is provided with a proprietary application. While none of the above-mentioned approaches represents a noticeable impediment in the case of single-sensor projects; the more sensors an application requires, the higher is the integration development effort. Also, interfacing devices from multiple manufacturers most often discards the possibility of using their proprietary software.

Previous semester projects in the LIS [4] [5] studied these sensors and identified the most commonly used ones. Most of these devices have an analog output or use data-link protocols such as I2C, UART, SPI or RS-422. Further a semester project [6] was launched to build a first prototype of a platform interfacing those standard sensors. Its aim was to be used in the prototyping phases of new projects and eliminating the need of hardware programming to interface those sensors.

The aim of this work is to reduce the general integration effort of sensors in future projects; making a step towards a unified framework for wearable technology in contrast to existing solutions like Arduino [7]. Ideally, the user should be able to handle in a simple and quick way to interface with hardware devices, regardless of their nature or supplier. To this goal, plug-and-play and hardware recognition functionality should be provided, changing the process of sensor interfacing to something similar to plugging a mouse in an USB port.

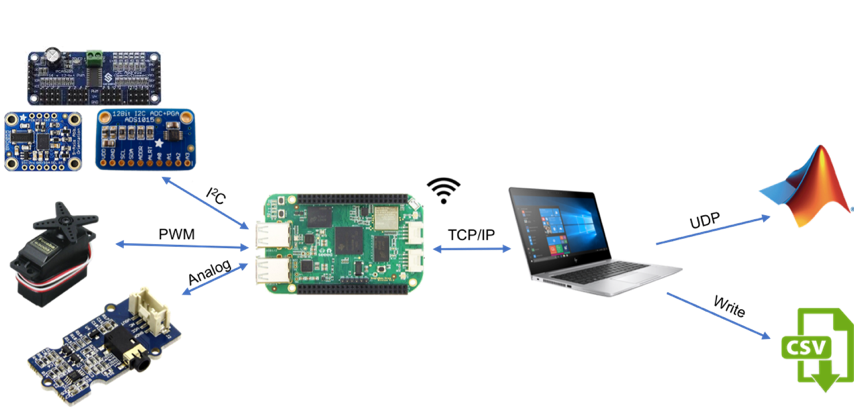


Figure 1 – Layout of the Plug’n’Play Framework

# State of the art

Since this project is aimed at developing a hardware-software middleware, it is only normal to first inspect the already available state-of-the-art solutions. Various such frameworks exist with different purposes; e.g. graphic engines such as OpenGL [8] or software frameworks such as the .Net [9]. In the following sub-sections, we will talk about the Robot Operating System (ROS) as one of the most widely used middle-ware not only for robotic systems and about the way normal Windows computers handle the connection of a new peripheral device such as a mouse or USB camera.

## ROS

The Robot Operating System (ROS) is a set of software libraries and tools that helps its users to build robot applications. It includes drivers, state-of-the-art algorithms and development tools [10]. ROS integrates the multitude of existing sensor and actuator hardware, often hard to use and incompatible by converting their data streams into a message bus, with compatible datatypes between the hardware drivers and calculation units. It also contains a set of conversion interfaces to run several external open source computation algorithms.

However, the great number of features – the vast majority of which are out of the scope of this project – represents a downside due to the increased complexity. The verbosity and high degree of formalism of ROS impose an unfriendly learning curve. Also, ROS is a purely software middleware. Thus, the hardware issue would still remain as it is not very well adapted to Linux systems.

## Arduino

Arduino [7] is a platform for low level hardware interaction between the Microcontroller *Arduino* and basic input / output devices like servos, analogs, I2C, SPI, etc. It provides drivers for a large range of devices, but requires programmatic interfacing from the drivers to a computer for further data processing. Therefore, prototyping and data acquisition is time consuming especially for people unexperienced in low level programming.

## LabView

LabView [11] is an industrial standard aimed at providing a visually manageable hardware / software platform for fast high-end data acquisition from a large range of hardware devices. It has a modular hardware architecture and a lot of preconfigured devices and services which can connected. However, it comes with a large initial cost. For any prototyping application that would require only a small sub-set of sensors and services, using LabView would be normally an overkill. A big downside of LabView is, it works only with proprietary devices.

## Universal Windows USB Drivers

In the introduction we gave the example of the act of connecting a new device to a PC via the USB interface. It is a process that encapsulates both hardware detection and plug-and-play functionality. Most USB devices need little to no configuration and work out-of-the-box when connected to a standard computing machine.

The USB bus is designed at the physical level so that the act of inserting or removing a device can be recognized by the host. When a connection event happens, the host controller scans the bus and asks each device to identify itself. All USB devices contain a collection of information about the device, called the descriptors. Device descriptors are retrieved from all devices with the same command, and this allows the USB host to effectively ask a newly connected device what it is, and expect to get a reasonable response.

Once the device is properly identified and classified into one of the standard USB classes, the system driver for each device class is usually sufficient to handle any devices that claim to be in that said class [12]. If not, drivers can be installed for the identified device from a remote location or even from the device itself.

While this mechanism works great for USB devices, it is impossible to implement as it is. This is due to the lack of standard connection events and “who are you” commands of the I2C, UART and SPI peripherals. Without these key pillars, the entire mechanism crumbles.

# What is already there?

This semester project continues another semester project from Gabriel Neamtu [6] which built the base of a Plug’n’Play platform.

## Hardware & Protocols

The project consisted in an evaluation determining the hardware fitting the problem the best, which was identified in the BeagleBone Green Wireless (BBGW) [13], since it has a large range of possibilities to interface hardware devices via I2C, SPI, UART, Analog, etc., while running on a UNIX operating system. Therefore a standard TCP/IP connection for development and data acquisition can be used.

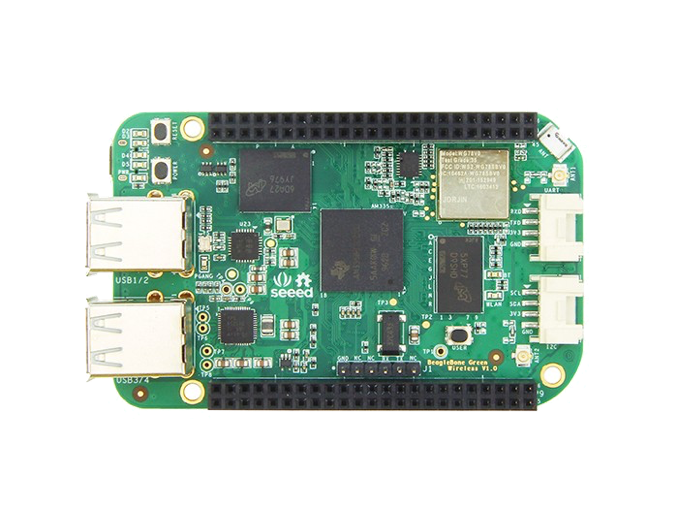


Figure 2 – Beagle Bone Green Wireless (BBGW)

## Software

The project provides a prototyped implementation of some of the previously mentioned protocols as well as a scanning algorithm allowing basic Plug’n’Play functionality. Further a connection protocol between the BBGW and a PC has been defined allowing fast and easy-to-use data acquisition via standard services.

# What is new?

A bundle of a Firmware and an Interface allows data acquisition and device control without to need to code on the Microcontroller. Instead, the visual Interface can be used to have live feedback and data streams. Further, an API allows third-party software the interact with the Framework to read sensor data or create closed loops.



Figure 3 – Interface used as an abstraction of the Firmware on the BBGW and the connected devices

# Project Structure

The project is divided into two parts, on one hand the Firmware on the BBGW and on the other hand the Interface on the PC. Both are coded in Python [14], the Firmware in python2, as a lot of hardware services are not yet available in python3, but for the Interface on the PC, python3 has been chosen.

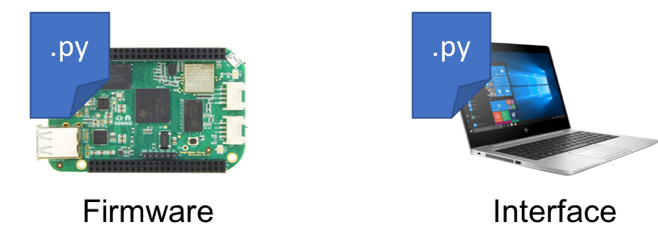


Figure 4 – Firmware is installed on the BBGW and the Interface runs on the PC

### GitHub

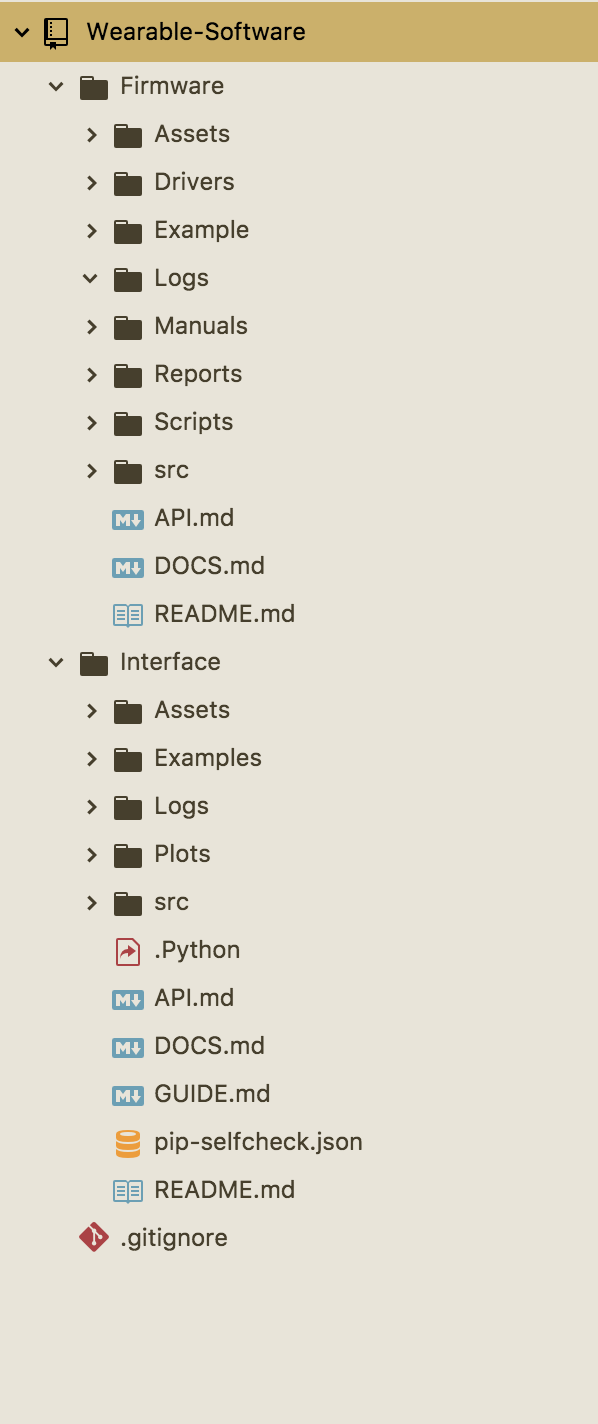
The two applications are managed and versioned in GitHub. Both parts have a similar but separated folder structure, containing a *README.md, DOCS.md* and *API.md* file (and a *GUIDE.md* file for the Interface) documenting the application, the installation, the API usage and a Guide about how to use the Interface. They also have a *src* folder with the files of the applications.

Figure 5 – Project folder structure on GitHub

For the Firmware, a folder *Drivers* provides the newest set of drivers required to connect the BBGW to the PC. Further, the folder *Scripts* contains scripts enabling/disabling auto-start of the Firmware. Last but not least, the folder *Example* provides sample code which can be used as a start point for new projects. Another *Example* folder can be found in the Interface.

### Docs

Documentations and Guides for the Firmware and Interface are provided along the applications giving an overview of how to use the Firmware or the Interface programmatically or via the GUI. One can either directly use the API of the Firmware to replace the provided Interface or let the Interface stream the data to a file for post-processing or via the UDP stream for real-time data acquisition. Live data visualization can be found in the Interface as well as settings for the various connected devices.

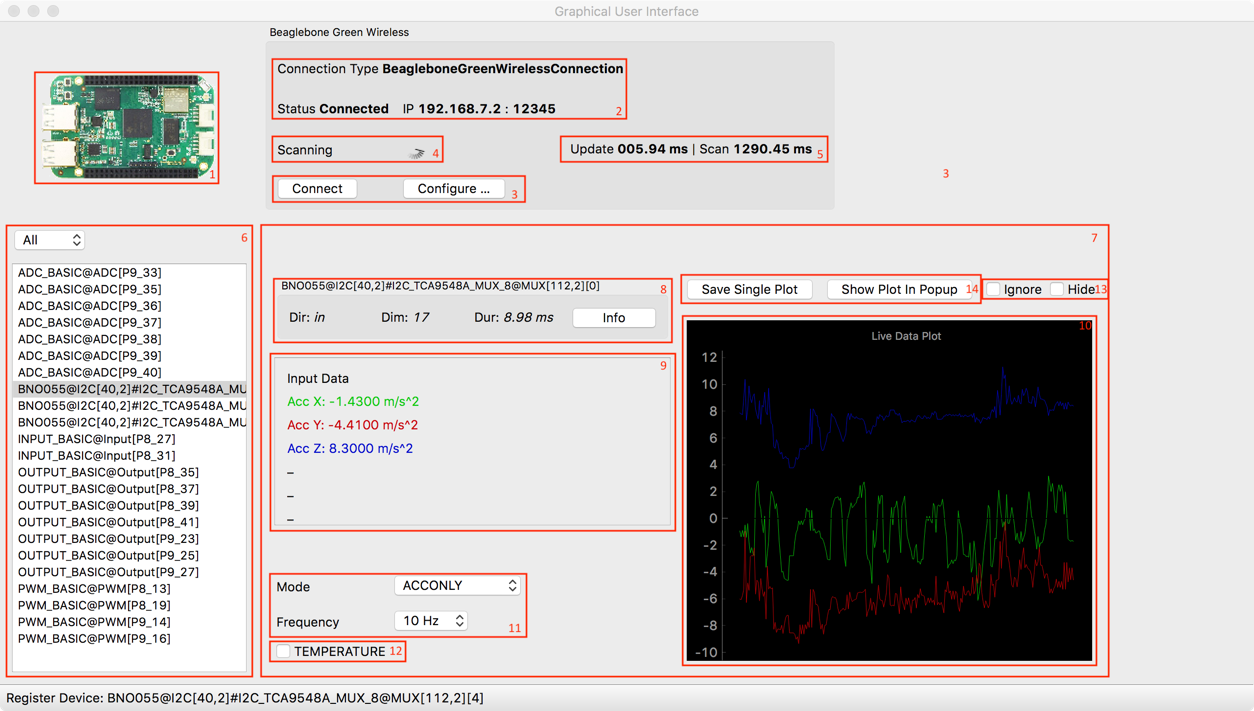
### Setup & Installation

The *README.md* files provide a detailed explanation of the installation and setup process of the Firmware on the BBGW and the Interface on the PC.

# Interface Architecture

The Interface is a visualization of the connected BBGW communicating with the installed Firmware via the API. PyQt5 [15] is used to as a widget framework and PyQtGraph [16] to draw plots in the windows. This Interface is just one of many possible ways to interface with the Firmware via the API and is an easy and fast way to prototype and debug projects.

A more in-depth explanation of the functionalities of the individual source files is presented in the documentation on GitHub. Further, a guide of how to use the GUI from a user perspective can also be found on GitHub.



Interface ∨

Board ∨

Help ∨

Figure 6 – Main Window of the Interface

## Main

The Interface provides a GUI for the Firmware on the BBGW using the API to receive and send data and messages. The main loads the board configurations, initializes the connection to the BBGW, launches the GUI and manages global events.

## Boards & Communication Methods

The configuration for the BBGW (#1) and its connection (#2) used by the Interface is stored in files, which can be chosen from and then used to connect to the selected board. This means, it is not limited only to the BBGW, but one can also implement other types of Boards with a similar API. Those files also serve as data models, storing all the run-time data used to display and plot in the Interface.

## GUI

The GUI consists of three main parts: the board, the devices, and the device information with the plots. The visualization is managed by the library PyQt5 [15], which is heavily event driven. The GUI is updated continuously when new data has arrived and is able to also send configuration messages via the API.

### Board Overview

The board overview shows which board (#1) is currently connected, the status, the IP and Port (#2), and update and scanning durations (#4 / #5) as well as the connection and configuration options (#3). A diagnosis plot of the update cycles of the Firmware can be shown from the *Board* menu.

### Device List

The list of currently connected devices (#6) can be filtered by device type and also ignored or hidden by raising the corresponding flag. *Hide* (#13) will remove it from the list and all plots where *Ignore* (#13) only removes it from the multi plots.

### Device Settings

The devices settings (#7) provides an overview of the currently connected and selected device (#8). Meta information as well as the live data values (#9) are displayed and when available, the functionality to set values and change settings (#11 / #12) is shown. There is also an option to stream the data for the selected device to a .*csv* file (#14).

### Live Data Visualization

Live data visualization is implemented in three different ways. On can directly see the live plot for a device by selecting it in the device list (#6). To view several devices together, one can either use the *Plot in Popup* (#14) option for the desired devices showing one device plot per window or by white- or black-listing the desired devices with the *Ignore* (#1) flag and use the *Multi Plot* option in the Board menu to plot the devices in the same window.

The live plotting is implemented using the library PyQtGraph [16].

## Communication to third-party software

The Interface provides three different types of data streams. It streams to *.csv* files for single devices from the device settings and for a set of devices from the Board menu. The set of devices can be configured with the *Ignore* flag similar to the *Multi Plot.* The format of the *.csv* file is documented on GitHub.

CSV

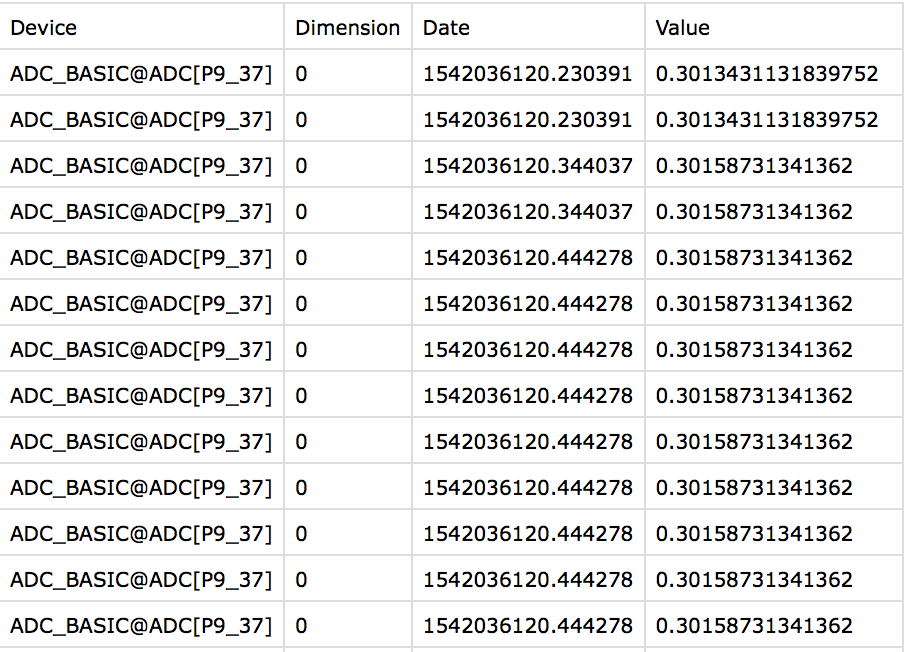
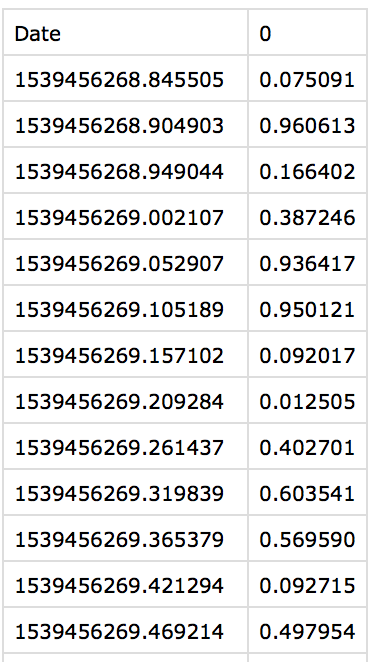


Figure 7 – Left: Single Device CSV Plot, Right: Multi Device CSV Plot

Another option is to let the Interface restream the received data messages via a UDP socket to a different port on the same PC. The instruction manual for the UPD stream can be found on GitHub.

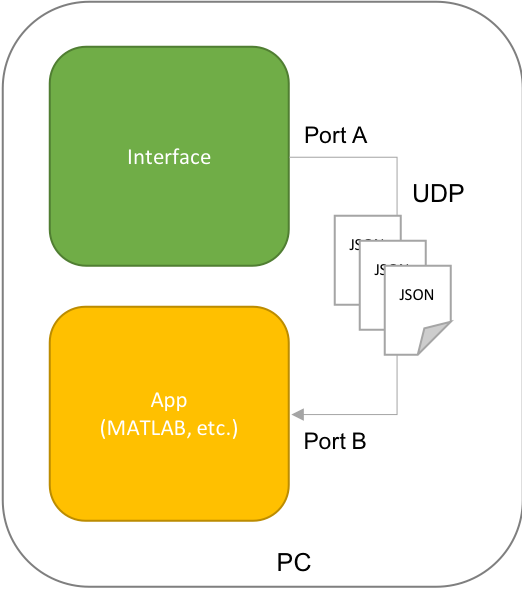


Figure 8 – Data stream from Interface via UDP to a different Port on the same PC.

## Diagnosis

The Interface can display a diagnosis plot of the update duration of the Firmware. Further the Interface can be run in debug mode.

# Firmware Architecture

The Firmware connects the physical devices with its drivers to an API which can be interfaced from the PC. To do so, it makes usage of a hardware library called Adafruit [17] which handles low level control of the I/O pins, the analogs and the data-link protocols like I2C, etc.

A more in-depth explanation of the functionalities of the individual files is present in the documentation on GitHub.

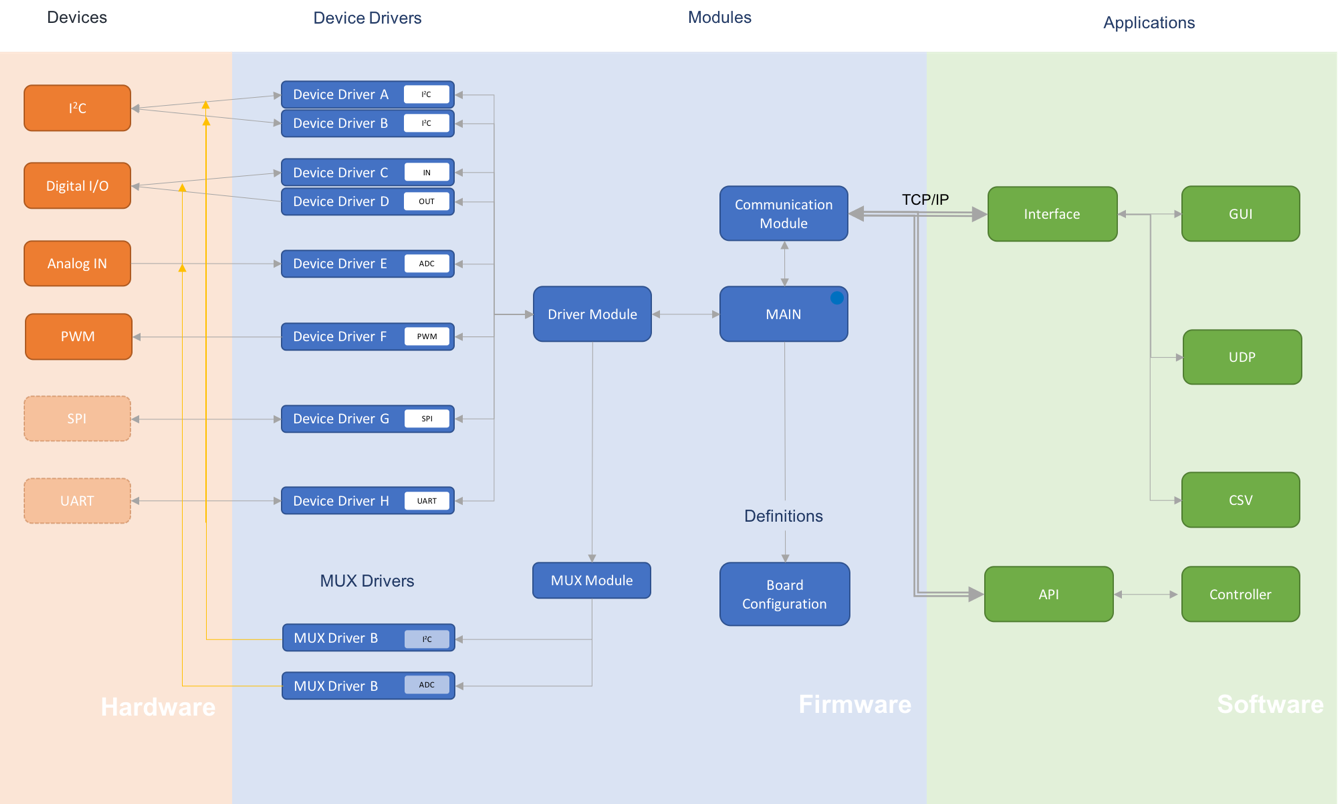


Figure 9 – Firmware architecture with hardware devices and connected software applications

## Main

The Firmware has three main tasks it needs to fulfil in parallel. The main takes care of them and acts as interface in between them. The threads run in parallel and are non-blocking, meaning the time to process is balanced in a Round Robin.

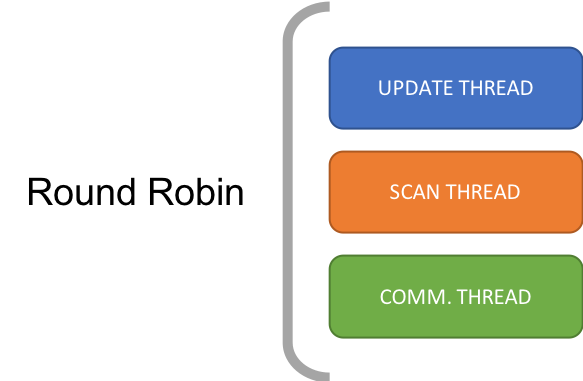


Figure 10 – Computing time between the threads is balanced with a Round Robin due to python

### Scan Thread

The scanning thread implements the Plug’n’Play functionality by looking for newly connected devices, checking for existing connections and cleaning up disconnected devices. This thread runs generally at a low frequency as it requires up to seconds to iterate through the list of all possible drivers for devices to perform a trial-and-error like device recognition. The effect is a laggy data acquisition for too many possible drivers. As a solution, it is possible to disable the scanning thread programmatically via the API or physically via a PIN on the BBGW.

### Update Thread

The updating thread gathers all new values from the connected devices. It is NOT performing the data acquisition, but only the transfer from the data stored in the drivers since its last iteration to the general data storage where it is kept in a standardized format for all devices ready to be sent by the communication thread via the API to the Interface or prepared for internal closed loop applications. The frequency of the update thread is in general as high as possible for the best possible performance. It does not run large computation as it only copies data and can be therefore run at high frequency.

### Communication Thread

The communication thread manages the connection from the Firmware to the PC and sends and receives messages via a TCP/IP connection. The frequency of this thread is usually adjusted to the network capabilities and requires only high speed for closed loop application with minimum delay. Else the data provides a timestamp allowing slower transmission without any loss in precision. The data provided by the update thread or coming from the API are wrapped into serialized JSON messages which can be sent as plain text or Buffer. Optionally, the communication thread will send the data to an onboard closed loop script.

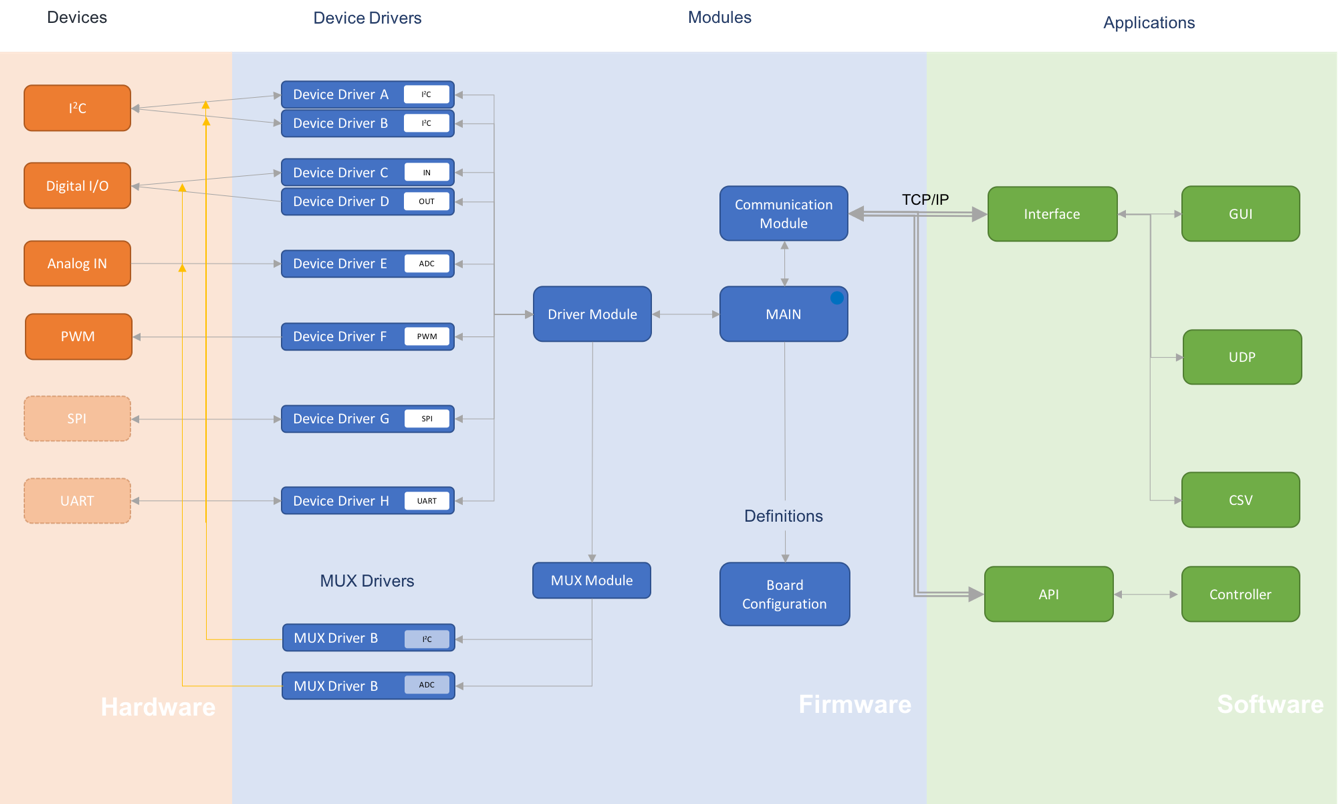


Figure 11 – Communication Thread of the Communication Module between the Main and the TCP/IP Socket for the Connection to the Software on the PC.

## Configuration

Two levels of configuration are implemented on the Firmware allowing basic Plug’n’Play configuration on a hardware level and advanced programmatic configuration on a software level to change the boards layout or the repartition of the different Pins and Addresses.

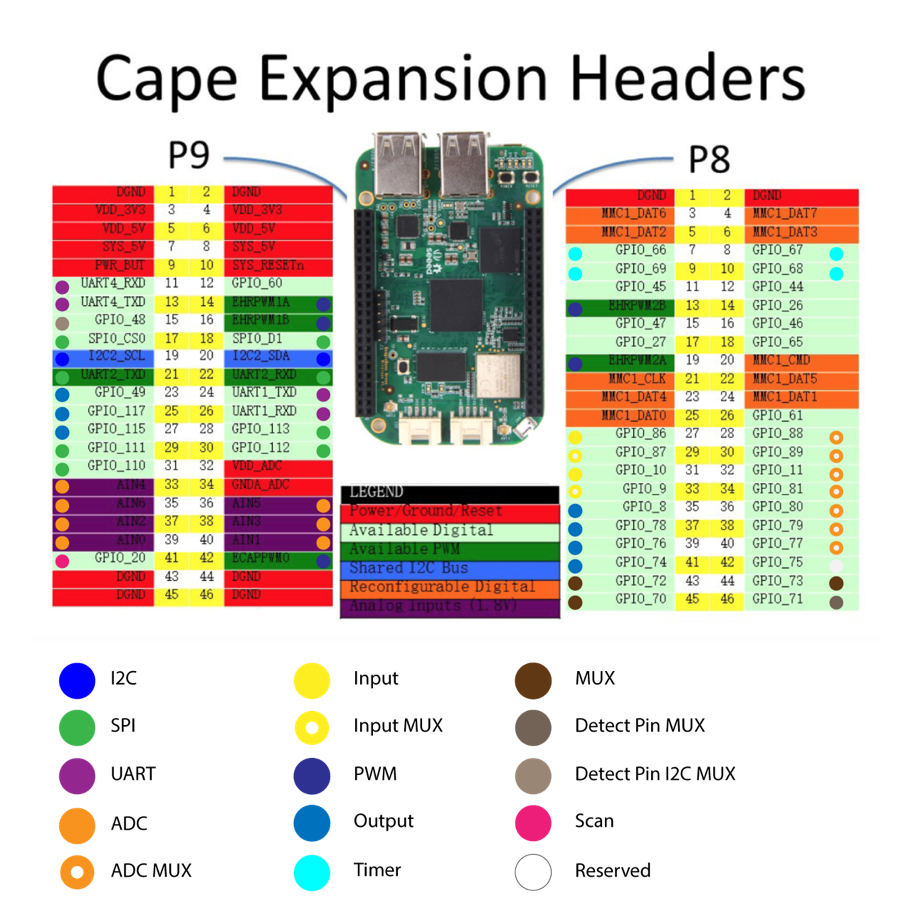
### Software Configuration

The software configuration file *Config.py* in the Firmware defines the layout of the Pins and the address assignments for the I2C devices. It provides a mapping between the implemented devices drivers and the hardware Pins. For example, the Pins for PWM could also be configured as basic digital I/O Pins if more are needed. As the I2C protocol bus provides a limited set of available addresses and some overlap cannot be avoided, one has to configure what driver is allowed on which addresses by hand.

Figure 12 – Software layout definition

### Hardware Configuration

The hardware configuration of the BBGW allows switching between different modes by setting special Pins to LOW or HIGH. Examples are the multiplexing for Analog Inputs, where one can connect a multiplexer to the analog input Pin by setting the corresponding configuration Pin to HIGH and disabling the scanning thread by simply setting a configuration Pin to HIGH or doing similar to enable I2C multiplexing.



Enable

Enable

Control

Figure 13 – Default Hardware Layout defined by the Firmware (The reserved pin P8\_42 can be used to extend the 8-Channel MUX control to 16-Channel MUX control)

## Drivers

The Firmware has a dedicated device manager for each protocol

* Basic Input
* Basic Output
* PWM
* ADC
* I2C
* *SPI (not implemented)*
* *UART (not implemented)*

Those managers are handling the list of connected devices of its type and map standard requests like *update*, *scan* and *settings* to the device drivers.

A special devices manager is the MUX device manager, as it is not used standalone. The other device managers use it to switch the MUX channels if they have a devices registered with MUX enabled. It maps requests like *scan, activate* and *deactivate* to the MUX device drivers.

### Device Drivers

Each device that should be detected by the Firmware, needs a driver which handles the communication to the device. The driver has a strictly defined set of properties and methods it needs to provide to be a valid device drivers like for example *identifier*, *data\_info, settings, check\_connection(), get\_values(), settings(), etc.*

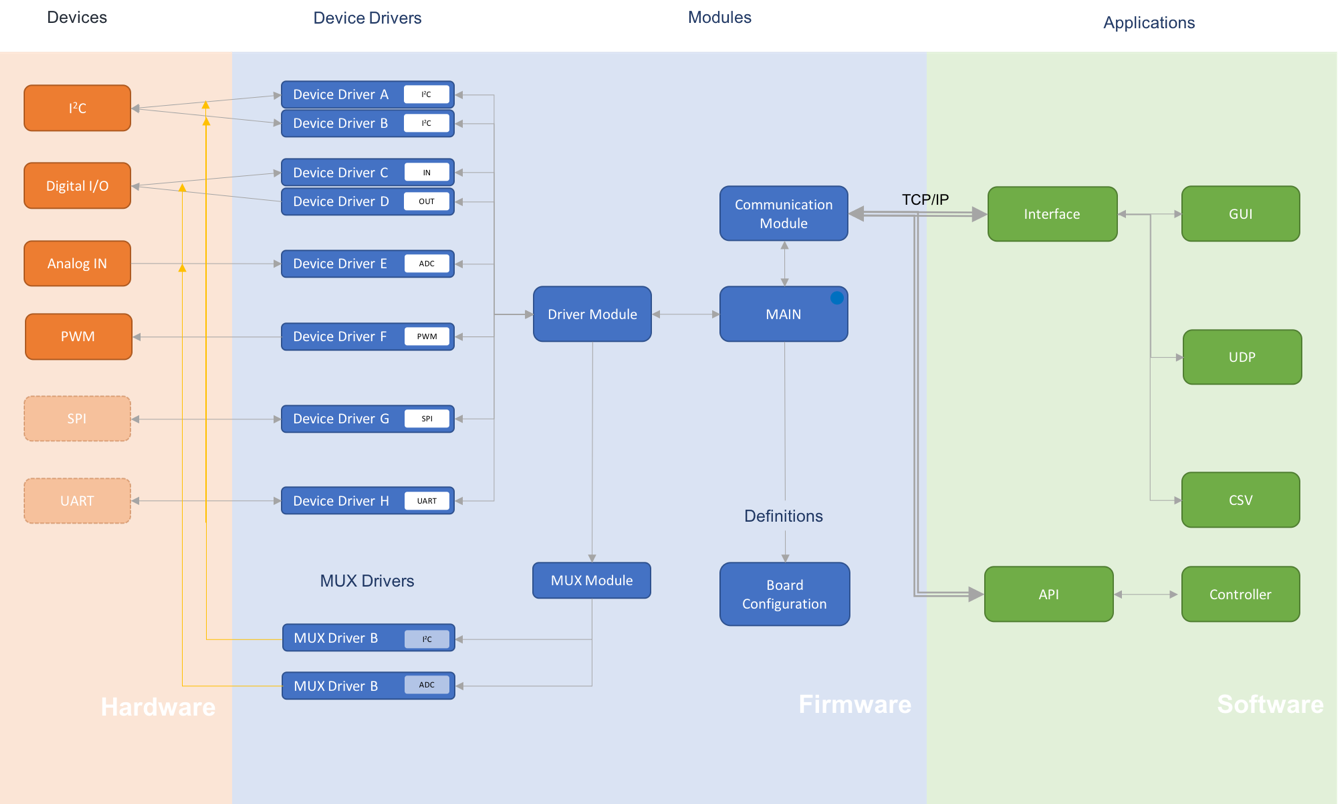


Figure 14 – Map between device protocol and driver implementation for I2C and Digital I/O for example.

For more detailed information about the requirements, a template is provided in the driver folder along the driver implementations.

Each driver has a dedicated and independent acquisition loop frequency to read and write data. This ensures optimal workload balance among the drivers and allows the high speed sensor drivers to take as much computation time as needed without being blocked by a basic digital input PIN driver. This frequency is generally individually configurable for each device via the API of the Firmware.

### Mux Drivers

The MUX drivers are different, as they have different requirements. In general, they do not need to read data and do not have a dedicated thread, but they provide methods to activate/deactivate a specific channel of the MUX device. For an analog MUX device, a set of Pins are used and for I2C MUX, the channel to set is sent via the I2C bus.

## Communication

For the communication, a standard library from python is used to create TCP/IP socket. The communication thread then sends and receives in a separate thread serialized JSON messages via this socket to the PC or directly to the embedded script files.

### API

To communicate with the Firmware from the PC, one can use the API of the Firmware on the TCP/IP connection to send and receive messages. A protocol to interact with the Firmware is implemented and the documentation can be found in the Firmware folder. The API allows to get data streams from connected devices, *register* and *deregister* messages for new devices, and send settings messages to configure the connected devices. An example of how to use the API on the PC is provided in the Example folder.

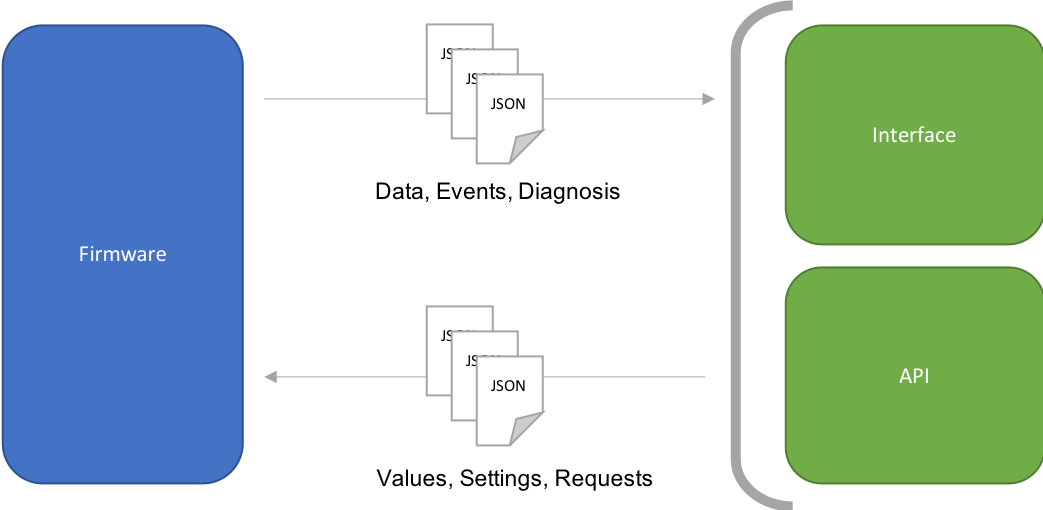


Figure 15 – TCP/IP Connection Protocol between Firmware and Software (Interface, API)

### Embedded Scripting

Alternatively, the embedded scripting, which is a specific implementation of the API, can be used to create a closed loop controller with only a few lines of code. An example of how to use it can be found in the Example folder. The difference between the API and the embedded scripts is, that the embedded scripts can be transferred to the Firmware to create an on-board closed loop without the delays of the TCP/IP connection. The programmatic interface for the loop to be created by the user is the same on the PC and on the BBGW, which means, one can develop and debug the script on the PC and finally transfer it to the Firmware. The embedded script is limited to one file named *loop.py*.

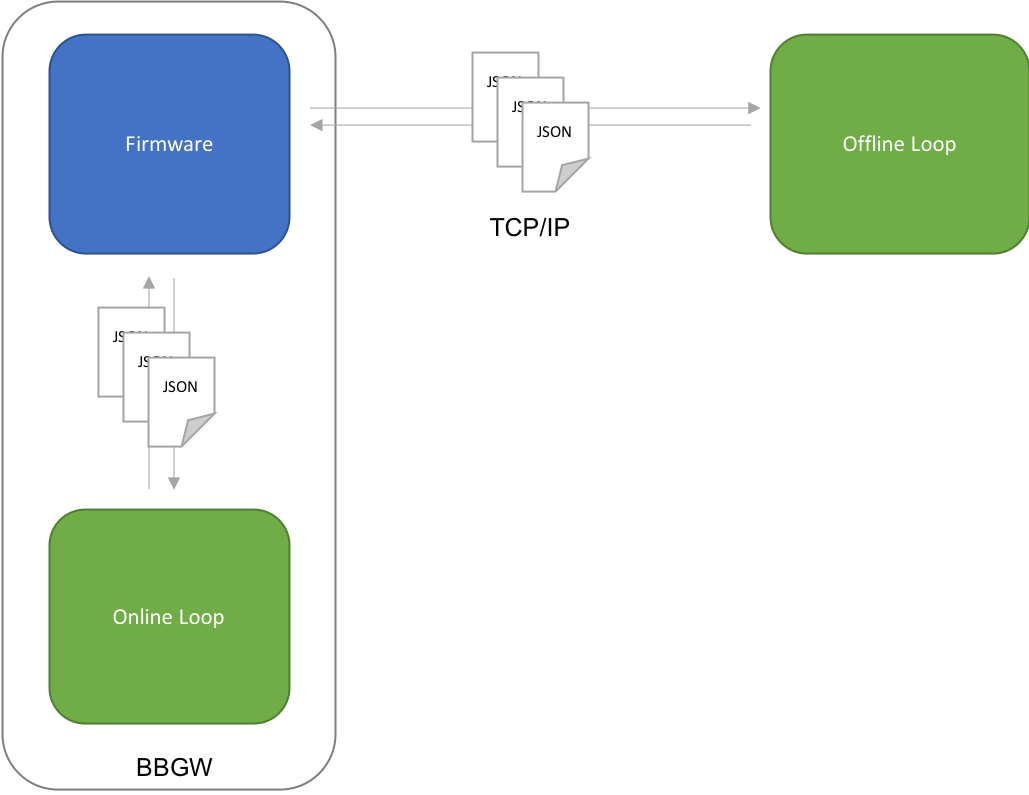


Figure 16 – Embedded scripting with closed loop online and offline.

## Diagnosis

The Firmware can be run with a diagnosis flag enabled, which will then make the Firmware log events, prints, and performance to the *Logs* folder.

# Demos

What can be accomplished with the Framework is presented with 3 small demos covering a large part of the available features like data acquisition, visualization and streaming, the modular driver structure and Plug’n’Play functionality as well as the API and Embedded Scripting.

## Data Acquisition, Visualization and Streaming

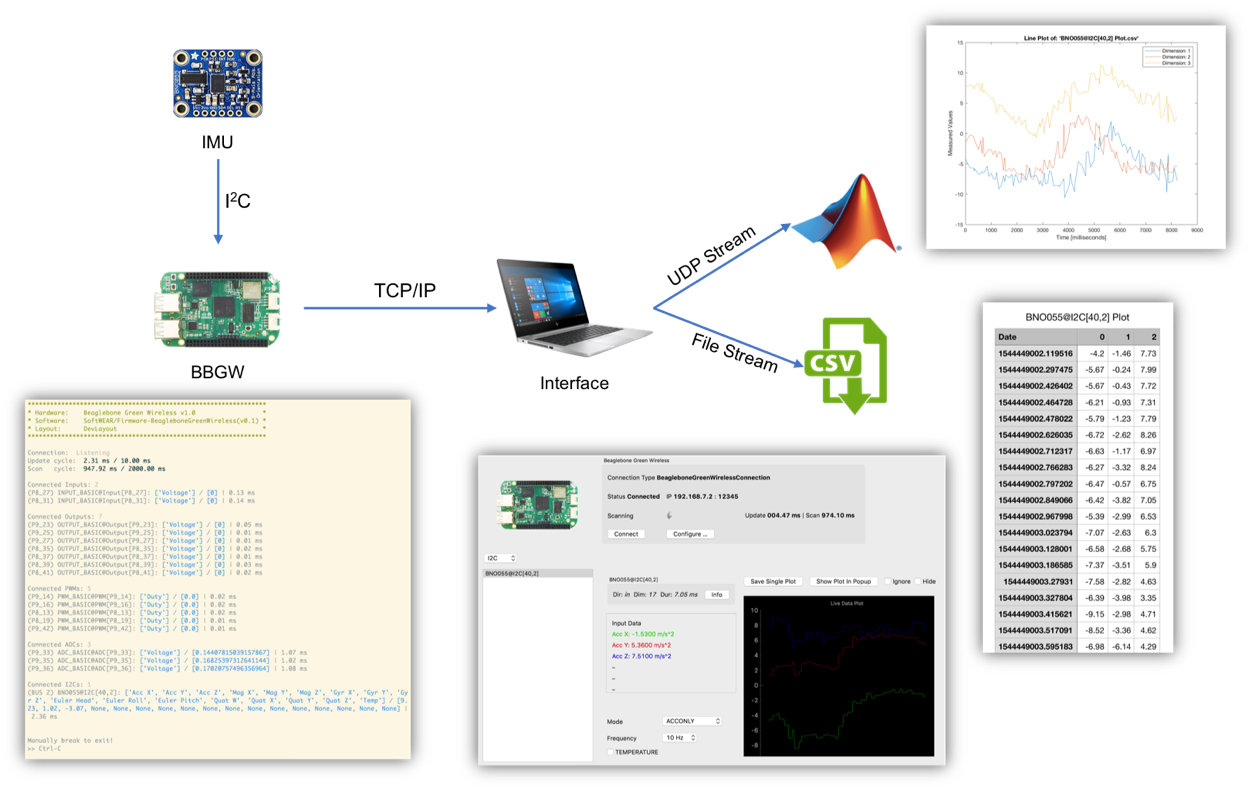


Figure 17 – Data acquisition, visualization and streaming of a sensor

A basic but essential task the framework can accomplish with a minimum effort is simple data acquisition. For example, an IMU can be plugged in on a BBGW while running the Firmware on board, and the device gets automatically recognized and listed in the console live feedback of the Firmware. Thus, the data gets sent to the PC where the next level is the Interface, which provides several possibilities of live data visualization plots either for a single device, or in case for multiple devices together. Further the Interface can stream the incoming data to a UDP Port for live processing of third-party software like MATLAB or Unity. For postprocessing, the data can be streamed to *.CSV* files.

## Plug’n’Play and Drivers

As the Firmware implements a Plug’n’Play functionality, the same Firmware can be used for multiple projects without changing code. With the appropriate drivers for the required devices installed, the Framework is able to interface those devices in an easy way as one can see in the following projects where none of students were required to code on BBGW to make their setups work.

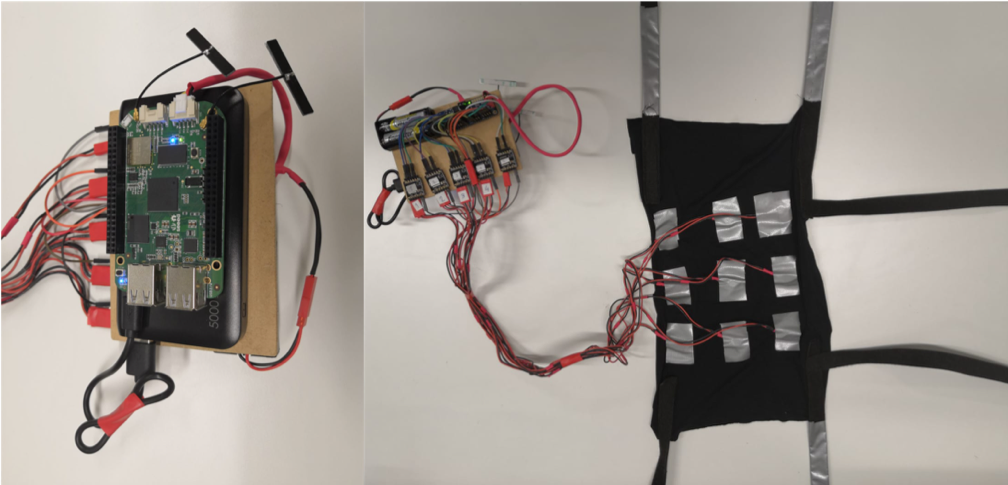
## API and Embedded Scripting

The API can be used for two different applications. Firstly, to connect a third-party tool like Hugo’s project or secondly, to create closed loops which can also be directly loaded onto the board.

### Semester Projects using the API

The API is already intensively used by the semester project “Haptic display for a wearable

human-robot interface” by Hugo Kohli [18], who is commanding an array of vibrating motors with a PWM multiplexer connected to the Firmware on a BBGW through the API.





TCP/IP

Figure 18 – Left: PC with pattern controller using the API. Middle: BBGW with Firmware connected to the PC with WIFI. Right: Motor array and PWM multiplexer connected to the BBGW with I2C.

The project “Hardware benchmark for an Inexpensive MoCap suit” by Victor Faraut [19] uses the data acquisition of the Framework to read out 9 individual IMUs by multiplexing the I2C Channel and streams it to a 3D visualization software.



Figure 19 – Sensor jacket with 9 IMUs and a BBGW with the I2C multiplexer shield

The project “Give the sense of touch to soft modular robots” by Siqi Zheng [20] uses a set of 6 analog inputs which can be read via the onboard ADC or a separate analog input device via I2C. The data is used to reconstruct poses of a tensegrity structure.

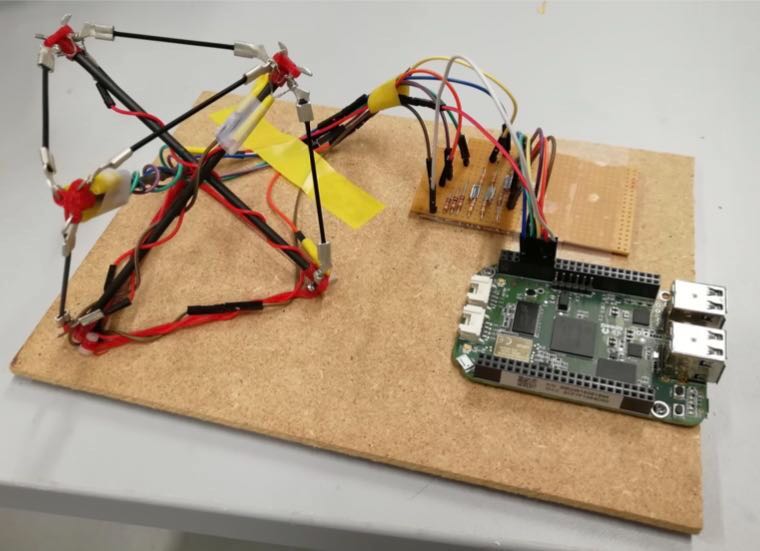


Figure 20 – Tensegrity with 6 analog sensors connected to the BBGW

### Simple Commands

With few lines of code, a simple loop can be implemented linking some buttons and analog inputs to an array of LEDs providing feedback. The script has an ON/OFF switch, a push button and an analog stick to get a basic three state value from the incoming update messages. The inputs are then mapped onto the LEDs with simple output messages.

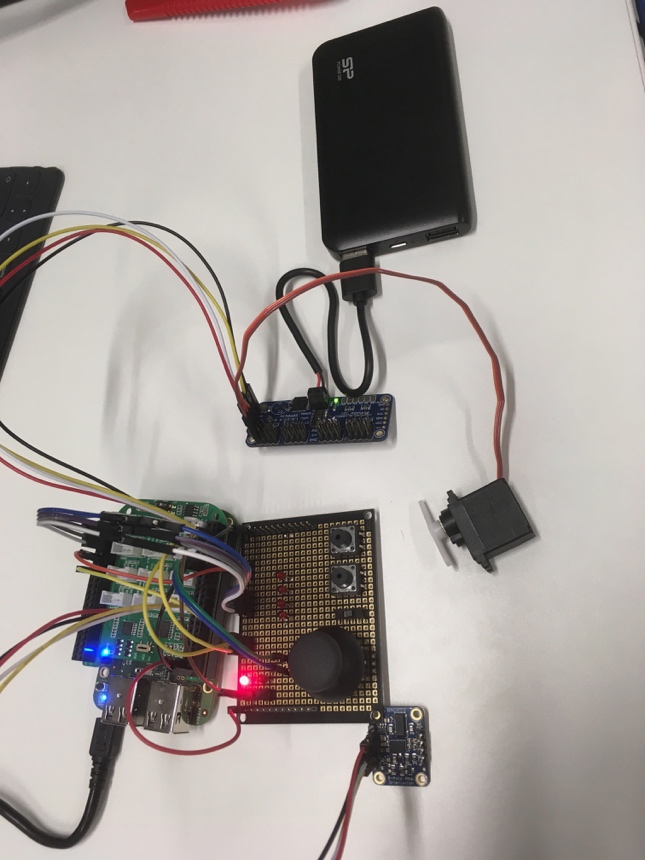


Figure 21 – Simple sample setup for a basic embedded script



39 Lines of Code

Figure 22 – Code to create the custom embedded loop script

### Connecting Two Sensors

With only **6** lines of code, a direct mapping from an IMU acceleration value to a Servo on a PWM Multiplexer bridge can be achieved. Depending on the acceleration measured by the IMU, the Servo changes its angle.

Compared to a similar system like Arduino [7], for untrained people several days would be needed to recreate the same functionality.

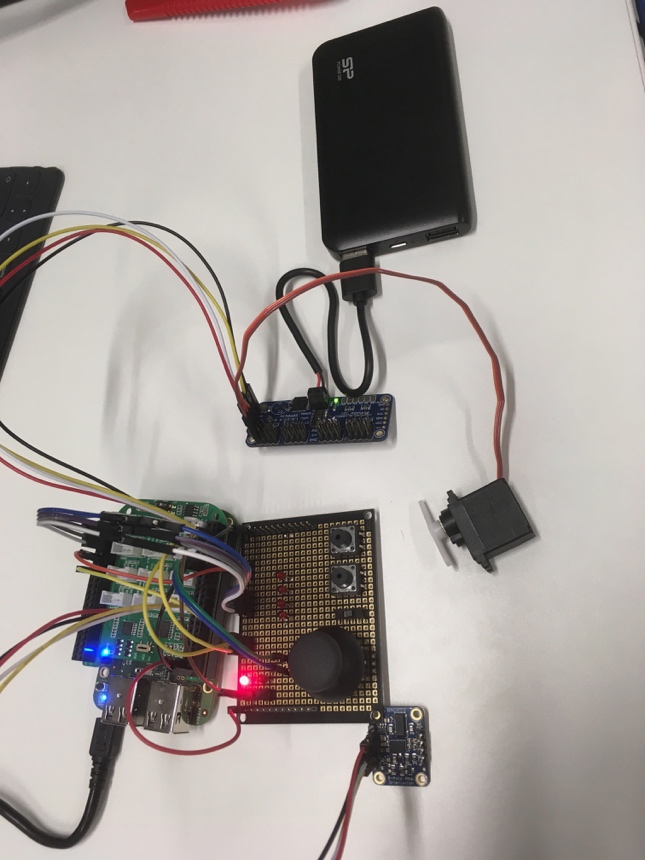
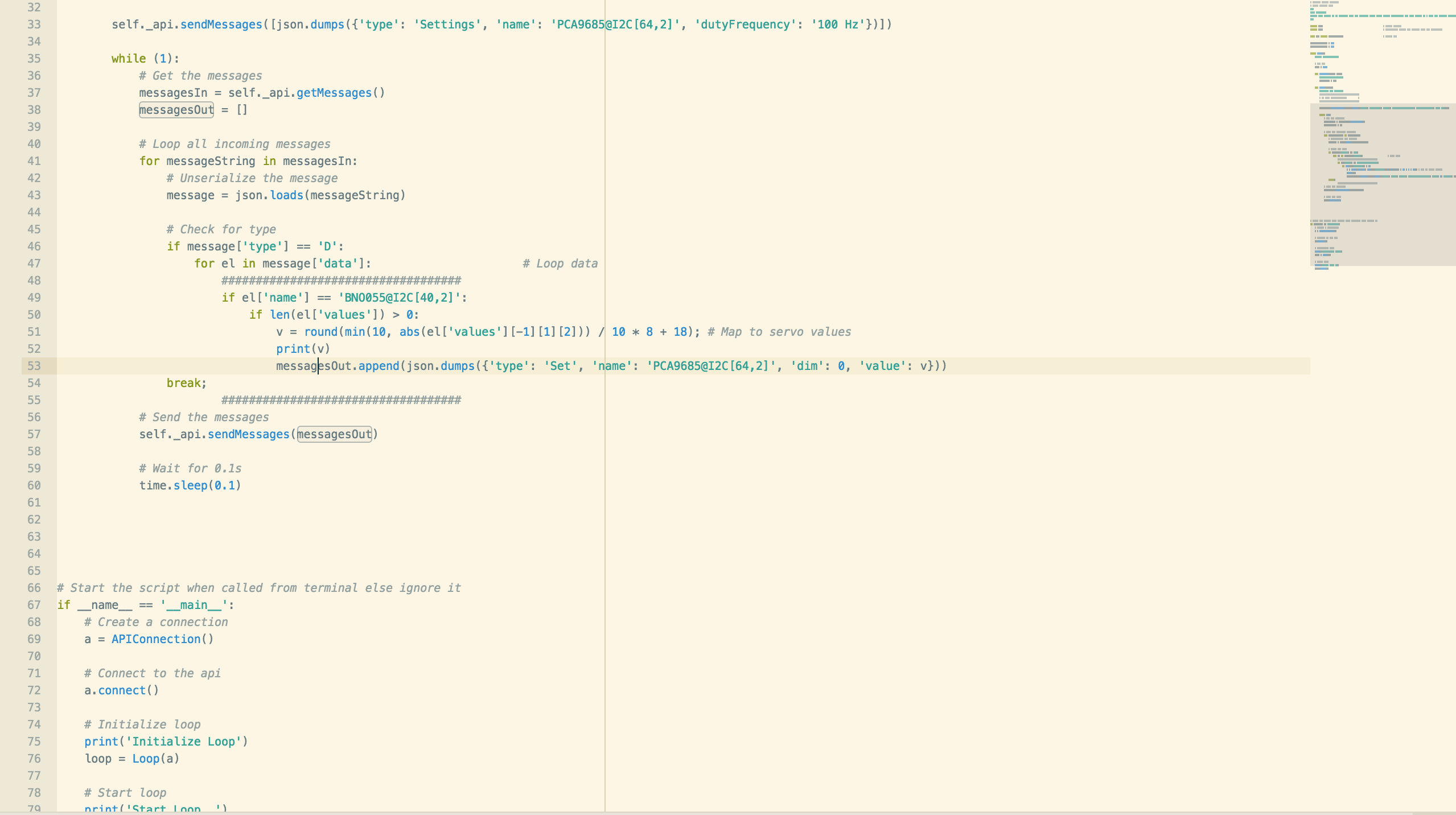


Figure 23 – Simple Sample Setup to connect two I2C devices with embedded scripting



6 Lines of Code

Figure 24 – Code to create a direct map between two I2C devices

## Diagnostics

Last but not least, the Firmware can be run in debug mode outputting a lot of diagnosis data covering performance, data acquisition, Plug’n’Play events and communication rates. Some of the diagnosis can even be streamed via the API to the Interface for live analysis while more in-depth logs are stored to log files to be transferred manually for analysis.



Figure 25 – Data Log containing all read values by the Firmware

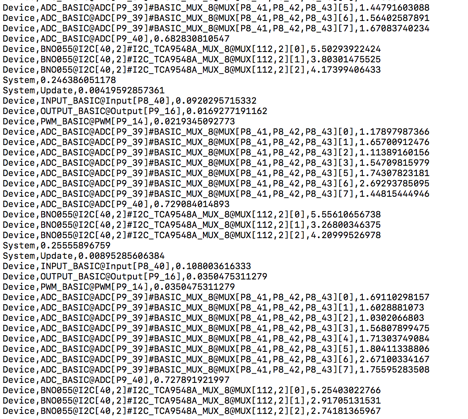


Figure 26 – Diagnosis Log containing the loop durations of the UPDATE and SCAN threads as well as the read durations of the different device drivers (This information can also be sent via the API for live analysis)



Figure 27 – Communication Log containing all the messages being exchanged via the API



Figure 28 – System Log of the BBGW containing for example CPU and RAM usage

# Evaluation

To evaluate the usability and performance of the framework, a feedback form has been handed out to the students who have been using this project for their own work. The overall feedback response is positive, what can be seen in the following results:



Figure 29 – 3 out of 3 students were satisfied with and would use the Framework again

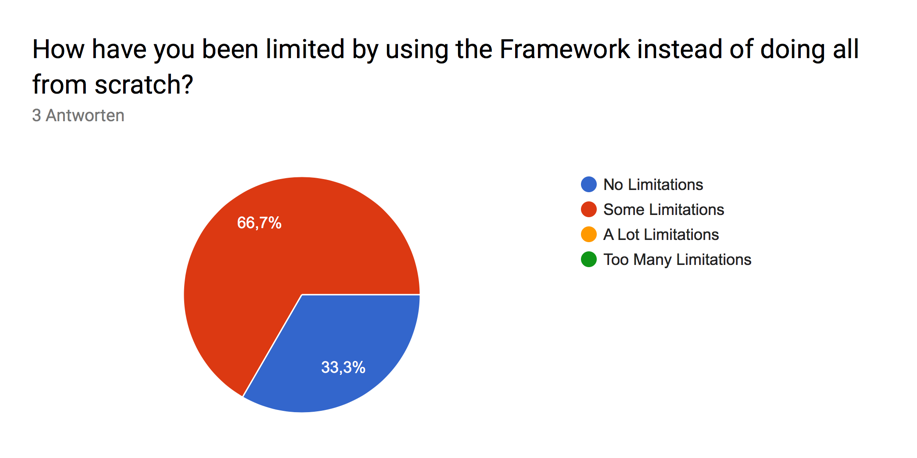


Figure 30 – The students were fairly limited in their projects by using the Framework

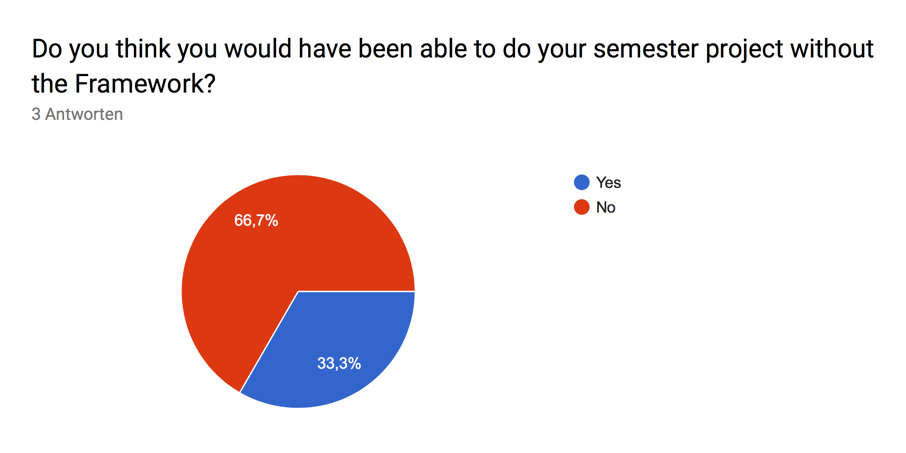


Figure 31 – 2 out of 3 students do not think they would have been able to complete their project without the Framework

As a wrap up, one can say the Framework successfully passed a first test as a prototyping Platform for students doing their projects. Further, the feedback of the students says that they saved up to 3 weeks of work by using the Framework. They were satisfied with the provided functionalities and would most likely recommend the Framework to other students.

# Future Work

Even-though a lot has already been accomplished, some further effort will be needed to finalize the project.

## Driver Installer and Custom Scanning

Currently, there are about 4 I2C drivers installed, which can be used to detect devices. As soon as there will be implemented more drivers, a severe problem will occur as the scanning cycle will take more and more time and start blocking other tasks like data acquisition. A first solution has already been introduced by the possibility to disable the scanning thread. But what happens when hundreds of drivers are installed. I2C for example has a very limited range of addresses and as soon as too many drivers are installed, collision is inevitable. Therefore one will need to introduce a possibility to enable and disable certain drivers at runtime. For example via the Interface, a list of drivers where one can check the drivers fitting the setup.

Further, as the Framework is developed and established, there might be even more drivers available from various sources, which will require a driver installer configuring the Firmware by downloading required drivers directly from the web.

## Embedded Scripting

The embedded scripting functionality allows online closed loop implementation by uploading a file *loop.py* to the Firmware with a script. There are quite some restrictions to the usage, like it needs to be one file and it cannot run independently. The scripts needs also to be manipulated directly in a python editor, which can be a rich source of bugs.

Therefore, to improve this functionality, a tool needs to be developed, which allows simple coding in python and takes care of the rest like the correct formatting and file upload.

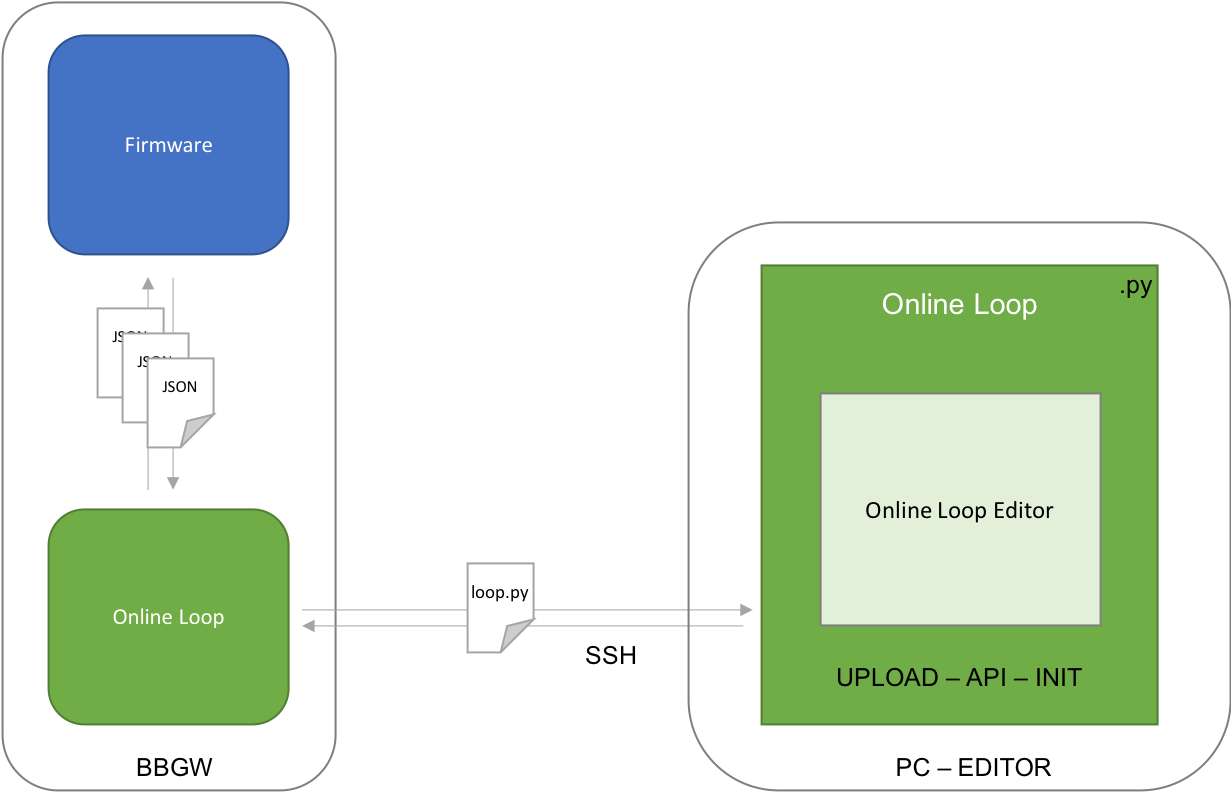


Figure 32 – Embedded Scripting Editor for online closed loop coding

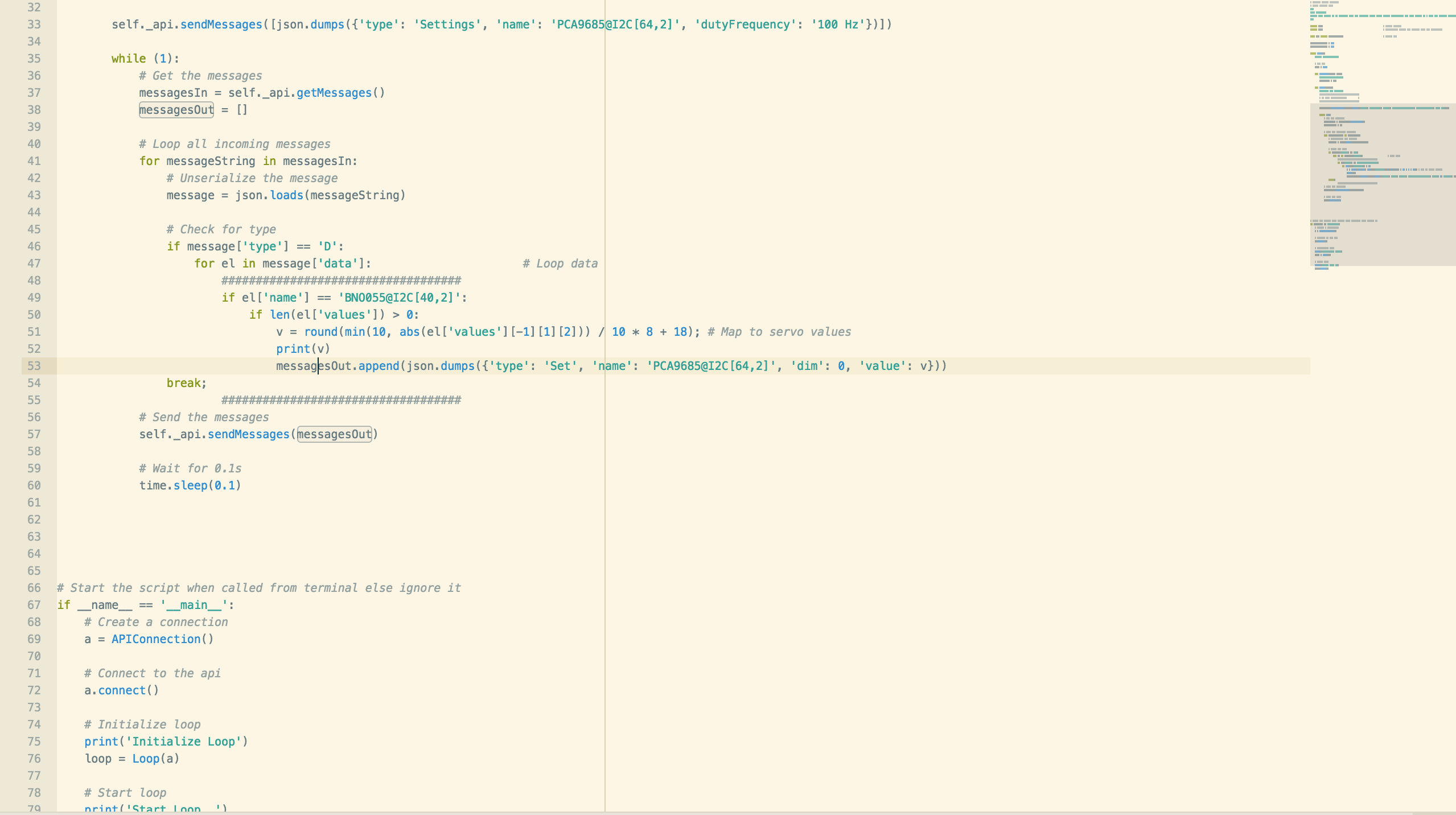


Figure 33 – Example loop.py file, where only the highlighted part is used for the custom loop and rest remains static across the different implementations

## SPI & UART

SPI and UART protocols are not yet implemented unlike I2C, which already has some drivers. Therefore, those protocols would need to be implemented as soon as they are required.

## C / C++

Python always has due to its nature of being a scripting language problems with performance compared to something like C or C++. To improve the performance of the Framework drastically, one would need to consider converting all the Firmware to C or C++. As the Firmware communicates with JSON messages via the API and does not depend on the programming language, on the PC side the Interface could still be used as well as every other third-party application.

# Conclusion

The end product includes all requirements stated at the beginning of the project. A standalone Firmware can be used for Plug’n’Play hardware devices on a BBGW. Further an API has been developed for communication, data streaming between the Firmware and computer software, and a standard Interface on top of tha API for prototyping. One can configure devices and read data without the need of hardware programming. Closed loops can be implemented by users with a simple workflow on high level programming instead of dealing with dedicated device drivers.

Overall, it was an interesting project and I have learned a lot about hardware programming on a microcontroller, the communication between such a device and software on a computer and data acquisition and treatment that is used for prototyping and research in labs.

# References

|  |  |
| --- | --- |
| [1] | H.-N. L. K. L. J. C. K. K. K. Vijayakumar Nanjappan, "Clothing-based wearable sensors for unobtrusive interactions with mobile devices," in *SoC Design Conference (ISOCC)*, Seoul, Korea (South), 2017. |
| [2] | A. A. F. R. S. M. D. A. Grégoire Surrel, "Online Obstructive Sleep Apnea Detection on Medical Wearable Sensors," *IEEE Transactions on Biomedical Circuits and Systems,* 2018. |
| [3] | S. M. F. D. A. C. D. A. D. F. Carine Rognon, "FlyJacket: An Upper Body Soft Exoskeleton for Immersive Drone Contro," *IEEE Robotics and Automation Letters,* vol. 3, no. 3, pp. 2362 - 2369, 2018. |
| [4] | H. Meyer, "A software for motion tracking and acquisition of biometric signals from wearable systems," 2018. |
| [5] | H. Johann, "Human body modeling and wearable sensors," 2018. |
| [6] | N. Gabriel, "Wearable Technology: Prototyping of a Modular Hardware Front-End," 2018. |
| [7] | Arduino, "Arduino," 2019. [Online]. Available: https://www.arduino.cc. [Accessed 2019]. |
| [8] | OpenGL, "OpenGL Overview," [Online]. Available: https://www.opengl.org/about/. |
| [9] | Microsoft, "What Is .NET?," 2019. [Online]. Available: https://www.microsoft.com/net/learn/what-is-dotnet. [Accessed 2019]. |
| [10] | L. S. E. F. L. J. Anil Mahtani, ROS Programming: Building Powerful Robots, 2018. |
| [11] | LabView, "NI – LabView," [Online]. Available: http://www.ni.com/de-ch.html. [Accessed 2019]. |
| [12] | Microsoft, "Getting Started with Universal Windows drivers," 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/develop/getting-started-with-universal-drivers. [Accessed 2019]. |
| [13] | BeagleBoard.org Foundation, "SeeedStudio BeagleBone Green Wireless," 2019. [Online]. Available: https://beagleboard.org/green-wireless. [Accessed 2019]. |
| [14] | Python Software Foundation, "python," 2019. [Online]. Available: https://www.python.org. [Accessed 2019]. |
| [15] | Riverbank Computing Limited, "PyQt5," 2019. [Online]. Available: https://pypi.org/project/PyQt5/. [Accessed 2019]. |
| [16] | Campagnola, Luke, "PyQtGraph," 2019. [Online]. Available: http://www.pyqtgraph.org. [Accessed 2019]. |
| [17] | Adafruit, "PyPI," 2019. [Online]. Available: https://pypi.org/project/Adafruit\_BBIO/. [Accessed 2019]. |
| [18] | H. Kohli, "Haptic display for a wearable human-robot interface," 2019. |
| [19] | V. Faraut, "Hardware benchmark for an Inexpensive MoCap suit," 2019. |
| [20] | S. Zheng, "Give the sense of touch to soft modular robots". |
| [21] | BeagleBoard.org Foundation, "BeagleBone Capes," 2019. [Online]. Available: https://beagleboard.org/capes. [Accessed 2019]. |