

EPFL

Laboratory of Intelligent Systems

Semester project II

SoftWEAR: Prototyping of a Modular

Hardware Front-End

*Responsibles:*

*Author:* Prof. Dario Floreano

Neamtu Gabriel Matteo Macchini

Anand Bhaskaran

Contents

[1 Introduction 3](#_Toc516175571)

[2 State of the art 4](#_Toc516175572)

[2.1 ROS 4](#_Toc516175573)

[2.2 Universal Windows USB Drivers 4](#_Toc516175574)

[2.3 BITalino 5](#_Toc516175575)

[3 General overview 5](#_Toc516175576)

[4 Board Selection 7](#_Toc516175577)

[5 Software Development 8](#_Toc516175578)

[5.1 Communication Module 8](#_Toc516175579)

[5.2 Hardware Detection 9](#_Toc516175580)

[5.2.1 ADC hardware detection 9](#_Toc516175581)

[5.2.2 Serial Hardware Detection 10](#_Toc516175582)

[5.3 Multiplexer Functionality 10](#_Toc516175583)

[5.4 Main Application 11](#_Toc516175584)

[6 Evaluation and Demo 13](#_Toc516175585)

[7 Conclusions and Further Work 16](#_Toc516175586)

[8 References 17](#_Toc516175587)

[Annex 1. Technical Documentation 19](#_Toc516175588)

[1. General Software Architecture and Software Unit interaction 19](#_Toc516175589)

[2. Known Issues 22](#_Toc516175590)

[3. Hardware considerations 23](#_Toc516175591)

# 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 wearable 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 its own driver, use an existing open-source one or, in some cases, is provided with a proprietary application. While none of the above-mentioned approaches represent a noticeable impediment in the case of one-sensor projects; the more sensors an application requires, the higher 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 as IMUs, EEGs and EMGs. Most of these devices have an analog output or use data-link protocols such as I2C, UART, SPI or RS-422. Thus, in addition to the difficulties in interfacing these sensors in the project software, one must also be careful about the hardware considerations.

The aim of this work is to reduce the general integration effort of wearable sensors in future projects; making a step towards a unified framework for wearable technology. Ideally, the user should be able to handle in a simple and quick way the interface with wearable 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.

# 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 [6] or software frameworks such as the .Net [7]. In the following sub-sections, we will talk about the Robot Operating System (ROS) as one of the most widely used middle-ware 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 [8]. 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, not the hardware issue would still remain.

## 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 connect 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 [9]. 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.

## BITalino

BITalino [10] is a similar project aimed at providing a hardware / software platform for wearable sensors used primarily in the medical field. It has a modular hardware architecture and aims at lowering the cost of sensor interconnect. However, the main weakness of this project is the very lacking peripheral capabilities. Any application that would require a larger amount of sensors would be incompatible with BITalino.

# General overview

As we have seen from the state of the art, solutions to similar problems already exist. However, we can only apply their architecture and principles of operation – as our exact application case is inconsistent with some of the features of the presented solutions.

As such, for SoftWEAR we would like a message communication system similar (but simplified) to the one used by ROS. Messages should be generated by sensors and hardware events. These messages should be sent to the PC and should also be freely available to any other application running in parallel with SoftWEAR. To implement this, the TCP/IP stack can be used due to its capability of sending messages over wired and wireless interfaces; and any parallel processes can use the loopback address.

However, the dynamic configuration that we want to create is cumbersome to implement in ROS. For hardware recognition, we can apply the universal Windows USB drivers’ principles. However, a method for detecting connections should be designed, as most peripherals do not have the same capabilities as a host USB controller. We will still use pre-developed drivers that will work when connecting supporting devices.

Our goal is thus to create a hardware-software middleware capable of interfacing with various sensors using analog output, SPI, I2C and UART. Both the hardware and the software should be modular and configurable. Due to the limited address space of I2C, multiplexer capabilities are also required. The plug-and-play feature should be available for both wearable sensors and multiplexers. Moreover, the connected hardware should automatically be identified.

The data hardware / software architecture and data flow of the end application is presented in Figure 3.1 below. The red arrow is the wired / wireless data link between the PC and the microcontroller board.

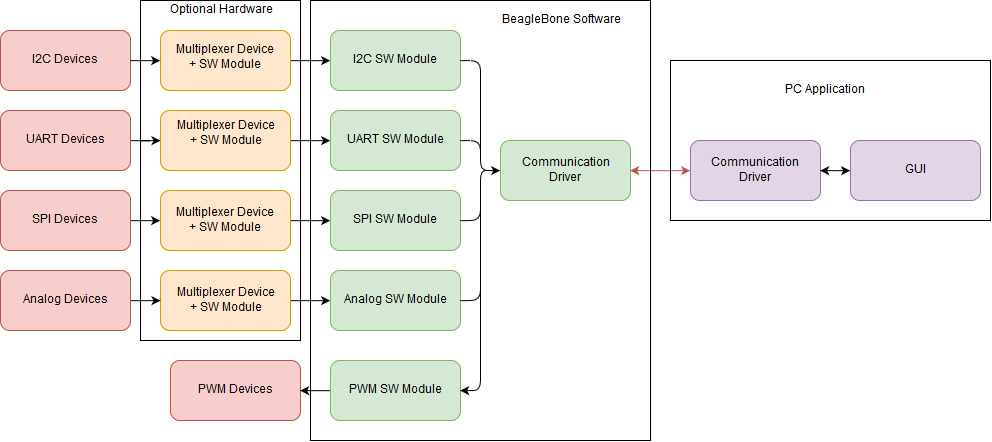


Figure 3.1. SoftWEAR hardware / software architecture and data flow.

# Board Selection

The first step in the project was selecting a suitable hardware platform. For reasons related to ease of development, a hard constraint was limiting the decision to existing development boards. Another crucial criterion was ease of usage, or LIS personnel familiarity with the hardware; as the time budget for dealing with the learning curve was limited. Thus, a short list of the most popular development boards and their main features is presented in Table 4.1. Note that this table also contains hardware that was at some point used by LIS members.

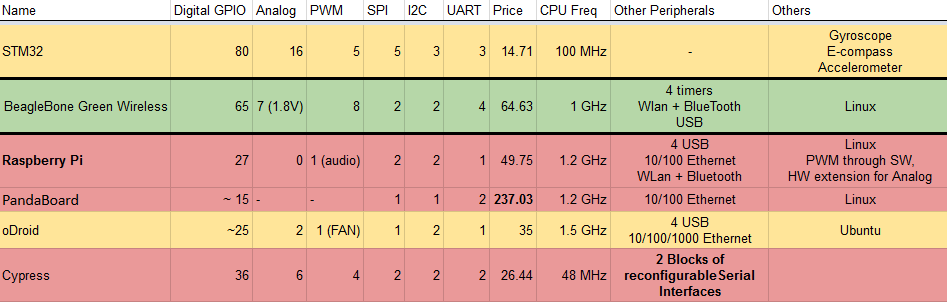


Table 4.1. Short list of the potential SoftWEAR hardware platform core

Table 4.1. also displays other soft criteria used for the final decision. These criteria are: price, peripheral capabilities, processing power and any pre-installed operating system. The final choice was the BeagleBone Green Wireless [11]. We considered this solution as the best compromise between all mentioned criteria.

Thus, as can be read from Table 4.1, our choice comes with pre-installed Linux that enables Python software development out-of-the-box. Also, the on-board WLAN and Bluetooth are key for a true wearable solution; while the rich peripheral capabilities completely satisfy all the hardware interfacing needs of the project.

# Software Development

The software is written to accomplish the tasks described in chapter 3; i.e. perform plug-and-play hardware recognition, communicate with a host PC, and add multiplexer functionality to the basic peripherals used by most wearable sensors.

The architecture of the project software is presented in Figure 5.1 below. It presents the individual units’ decomposition and interactions, as well as their implementation status.

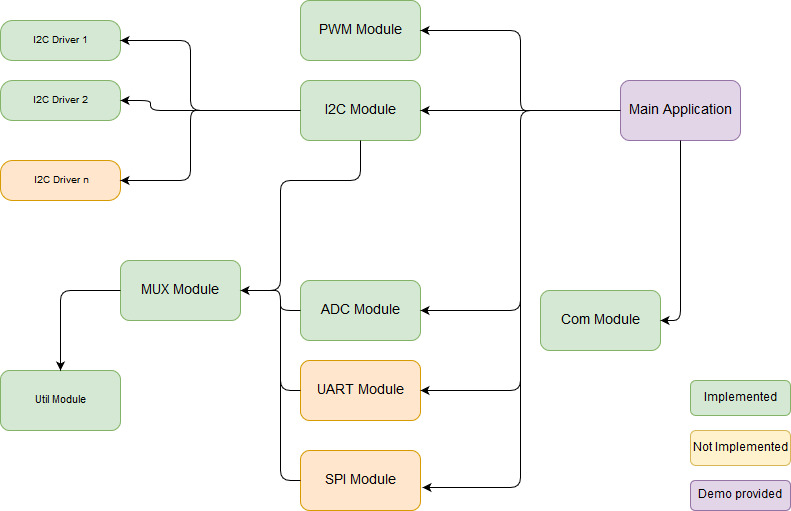


Figure 5.1. SoftWEAR software architecture. Not implemented software

units are out of the scope of this semester project.

## Communication Module

The communication module uses a message system principle similar to ROS, but greatly simplified. Peripheral modules communicate with the main application using a dictionary-based message format. The same method of passing information is used by the SoftWEAR communication class. Thus, each message sent is a python dictionary – facilitating the ease of processing by parsing and classifying.

The communication module is also built to facilitate features such as asynchronous non-blocking calls and automatic dictionary serializing and de-serializing. In order to implement these features, a background thread is created and all TCP/IP underlying sockets are running in parallel with any user application. The data synchronization is done by the use of two queues for sending and receiving process. The communication module architecture is presented in Figure 5.2.

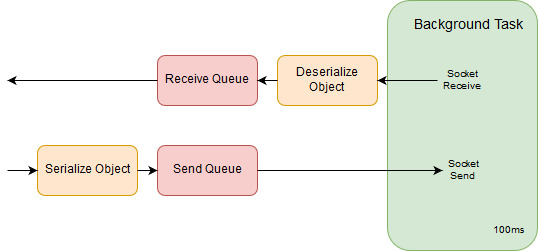


Figure 5.2. Communication module architecture

## Hardware Detection

The hardware detection is different from peripheral to peripheral and thus implemented in each class (ADC, SPI, I2C, UART) separately. It is responsible for detecting when a new device is connected and for correctly identifying it. It is also implements ubiquitous driver loading for each connected device.

### ADC hardware detection

In order to detect when a new device with analog output is connected the ADC channel is normally pulled to 0V via a pull-down resistor. This state is recognized as the ‘disconnected’ state. When a new sensor is connected, an analog signal is present at the respective channel. Thus, when the board software detects a signal above 0V for a set number of consecutive reads (for de-bouncing purposes), it will signal the connection of a new device. Similarly, when the signal is close to 0V for the same amount of consecutive reads, it will report the channel as disconnected.

The main setback of this approach is the lack of device model and type detection. However, this can be compensated by user dialog boxes; as EEG and EMG behavior is rather similar between sensor models. Thus, the number of user actions required should be relatively low and limited, among others, to model selection and ADC range (e.g. 0-3.3V).

### Serial Hardware Detection

In this section we focus on I2C hardware detection, but the same mechanism is applied for all devices using any of the serial protocols. The method is similar to the one described in subsection 2.2. However, because there is no unified command to ask sensors about their identification data, drivers are loaded and kept in a list before identification.

At module initialization all drivers are loaded and kept in a separate list for each hardware channel. Each driver has to implement a couple of required functions. One of them is called getDeviceConnected() which returns ‘True’ only if there is a very high certainty that the device is connected. A good way to implement this function is by reading from a device register that contains a specific read-only constant. The odds of two different devices to share the same address and read the same constant from the same memory location are 1 in 16 million.

These said driver function is called sequentially for all drivers in the list. When a supported sensor is connected, the call from its respective driver returns ‘True’ and the device is recognized and identified. While this method can scale poorly when applied to a large list of potential drivers, it has the advantage of not requiring any extra hardware. Thus, it represents a true out-of-the-box solution.

## Multiplexer Functionality

As mentioned in section 3, the multiplexer is required for a true plug-and-play solution – mainly due to I2C limited addressing space for identical devices. It is also useful in extending existing scarce hardware resources such as I2C channels.

In SoftWEAR, the MUX functionality is encapsulated in a separate class and used in all of the other peripheral classes except PWM. Every multiplexer object requires 4 reserved GPIO pins: the select pins (A, B, C) and the detect pin. The MUX has hardware detection functionality through the detect pin – thus adding a MUX to a channel can also be done in a plug-and-play fashion. See Figure 5.3. for the connection diagram of a multiplexer.

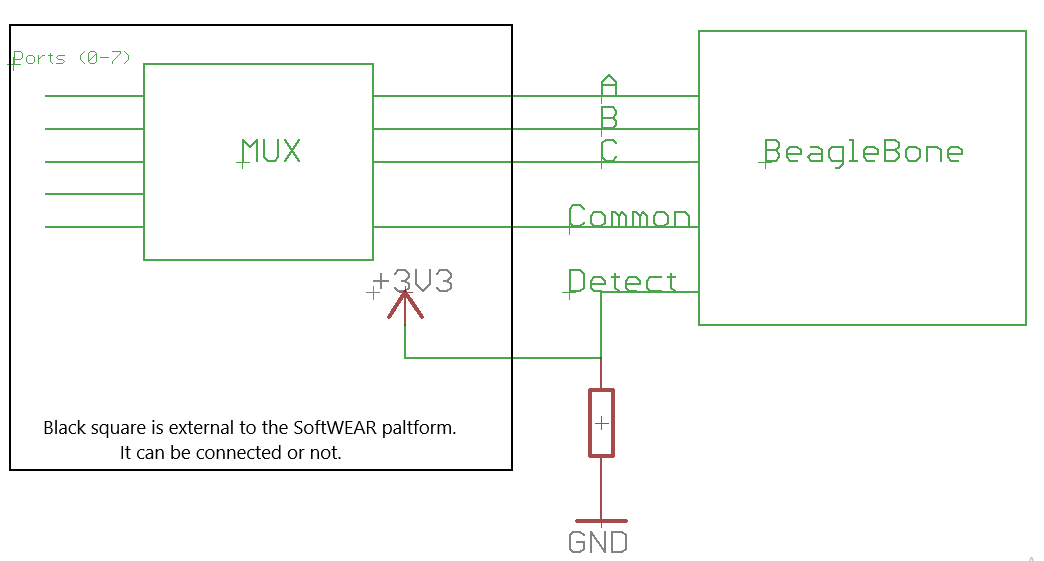


Figure 5.3. Multiplexer connection diagram. The Common pin is a peripheral pin.

Power lines not shown.

If a multiplexer is disconnected, the detect pin is pulled to ground through the pull-down resistor. Otherwise, the ‘1’ logic level is applied to the same detect pin. Using this simple mechanism hardware detection is implemented on the multiplexer. Once the presence or absence of a multiplexer is asserted - SoftWEAR either executes the intended peripheral operation once, or multiple times while changing the select pins after each call.

## Main Application

The main application is responsible for coordinating all SoftWEAR peripheral calls and send their results over the network. A main application demo is provided that uses ADC, PWM and I2C with all their underlying features. Ideally, the main application is configured to run at start-up. Also, all messages that are sent to the PC could be sent via loopback address for use in other potential running programs, as presented in Figure 5.4. below. The development of these embedded programs should greatly be simplified by the features offered by SoftWEAR.

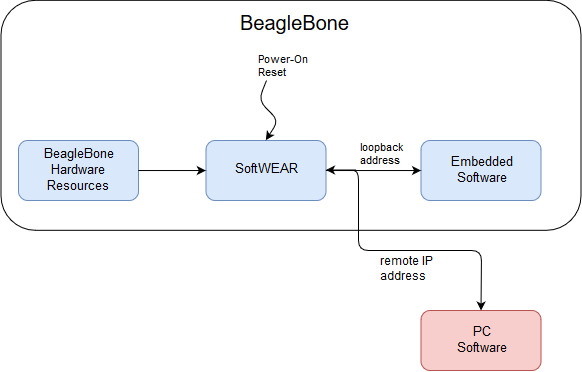


Figure 5.4. Adding Embedded Software applications running on top of SoftWEAR

# Evaluation and Demo

As mentioned in section 5.4, a demo application is provided that demonstrates the main features of SoftWEAR. It is comprised of both a PC Graphical User Interface and a SoftWEAR main application that runs on the hardware platform.

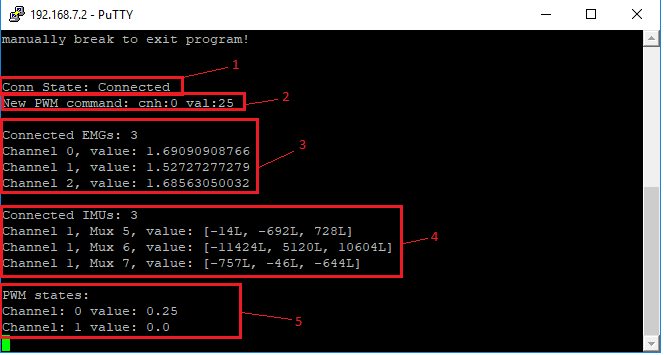


Figure 6.1. BeagleBone Software. Elements displayed: Connection state (1),

Last event (2), State of ADC channels (3), State of I2C channels (4)

State of PWM channels (5)

Figures 6.1 and 6.2 present the demo applications and their displayed elements. The required features are demonstrated, i.e. hardware detection, plug-and-play and multiplexing capabilities. Two I2C IMU sensors are supported: BNO055 and MPU6050. In the example 2 BNO055 and 1 MPU6050 are connected to the same I2C channel via a multiplexer. All information is displayed in the console output of the BeagleBone and on the PC software GUI.

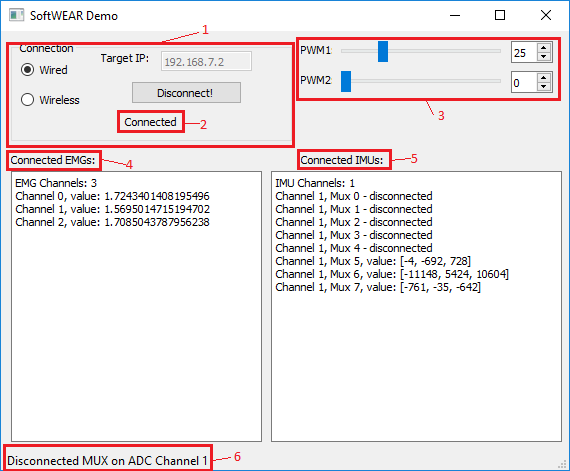


Figure 5.2. PC User Interface. Elements displayed: Connection window (1),

Connection Status (2), PWM controls (3), State of ADC channels (4),

State of I2C channels (5), Last event (6)

By using the SoftWEAR package to develop this demo, as expected, the programming effort was greatly reduced compared to a starting from scratch approach. But more importantly, the code was designed in a modular architecture that should support any new application – thus reducing the development effort for any future projects that fit the hardware constraints.

The code characteristics of the demo application are presented in Table 6.1. As we can see, the amount of code required to use the SoftWEAR (core code) is relatively low for the rich features it provides.

|  |  |  |
| --- | --- | --- |
| Application | Scope | Aprox. Lines Of Code |
| BeagleBone | Core Code | 45 |
| BeagleBone | Display Code | 100 |
| BeagleBone | Auxiliary Code | 40 |
| PC GUI | Core Code | 25 |
| PC GUI | Display Code | 120 |
| PC GUI | Auxiliary Code | 40 |

Table 6.1. Lines of Code in the PC software and SoftWEAR BeagleBone

application grouped by category.

# Conclusions and Further Work

The main objectives of the project were reached; i.e. a hardware-software middleware that has the potential to dramatically reduce wearable sensor integration time of future projects. A suitable hardware platform was chosen with respect to the mentioned criteria. Based on this platform, software was written implementing the desired features: wearable sensors plug-&-play, hardware recognition and detection, multiplexer capabilities and communication with the PC. Integration of several peripherals (PWM, ADC, I2C) is complete. A demo application was provided for feature display and evaluation purposes.

However, there is still unfinished functionality. Thus, the UART and SPI packages are still to be implemented. Also, a standard SoftWEAR main application that runs on start-up and provides an API for sending messages on loopback address is also required for a complete software project. Also, a list of current issues is present in Appendix 1. In addition to this, a hardware design is also desirable. Due to the nature of the software, this design can be modular and flexible and can be organized as a BeagleBone Cape [12]. It could include a power source and Wi-Fi configuration for true wearable functionality.

By exploiting the software configurability and flexibility, different capes can be designed for different application fields. Using the multiplexer feature to augment the available number of resources, capes can be designed supporting a large amount of sensors. For example, for applications that require complete motion tracking and using a large amount of IMUs, a cape with many I2C (e.g. 16) channels can be designed. Other applications for EMG / EEG requiring ADC capability could make use of a cape with a multitude of ADC channels (up to 56). These can be designed without any need to change the software due to the MUX hardware detection – just plugging the cape should suffice.

On top of this, the feature of on-the-board-control can easily be integrated by the using the principles presented in subsection 5.4. Thus, in addition to the interfacing role that SoftWEAR has, it can also become a true hardware-software development platform for future wearable robotics applications.

# 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] | OpenGL, "OpenGL Overview," [Online]. Available: https://www.opengl.org/about/. |
| [7] | Microsoft, "What Is .NET?," [Online]. Available: https://www.microsoft.com/net/learn/what-is-dotnet. |
| [8] | L. S. E. F. L. J. Anil Mahtani, ROS Programming: Building Powerful Robots, 2018. |
| [9] | Microsoft, "Getting Started with Universal Windows drivers," [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/develop/getting-started-with-universal-drivers. |
| [10] | Bitalino, "BITalino - Biomedical Equipment | Low cost Toolkit," [Online]. Available: http://bitalino.com/en/. |
| [11] | beagleboard.org, "SeeedStudio BeagleBone Green Wireless," [Online]. Available: https://beagleboard.org/green-wireless. |
| [12] | beagleboard.org, "BeagleBone Capes," [Online]. Available: https://beagleboard.org/capes. |

# Annex 1. Technical Documentation

## General Software Architecture and Software Unit interaction

The general architecture and the software unit interaction of SoftWEAR is presented in Figure 5.1 from section 5.

At the moment of writing, the UART and SPI modules are not implemented. Also, only two I2C device drivers are implemented – but support exists for any number of future implementations.

* 1. PWM Module.

There are 2 module dictionaries used to translate between readable indexes or pin identifiers (e.g. 0-7; ‘P8\_13’) up to the identifiers used by the mraa library. These dictionaries are constant snd should not be changed.

The main application should only create one class instance of the PWM module, as it encapsulates read / write functionality for all configured PWM channels. To configure a channel, simply add it to the RoboPWM.WRITE\_PINS list. This list SHOULD NEVER BE MODFIED AT RUNTIME. No operation will be supported for pins not present in this list. This is done for the purpose of providing the user with a list of usable PWM channels while still having the possibility of using some PWM hardware for built-in purposes.

* 1. I2C Module

All information presented in annex section 1.a are valid for the I2C module; with the exception of lacking the index translation dictionaries.

In addition to configuring the usable I2C channels that the user has acces to; each I2C channel has configurable Multiplexer (MUX) pins by using the list RoboI2C.SCAN\_CHNS\_MUX. Same as the SCAN\_LIST, this variable SHOULD NEVER ME MODIFIED AT RUNTIME. If no multiplexer operation is supported, fill the coresponding slot with ‘None’. If MUX features are desired, the corresponding slot should be filled with a list of digital pins [A, B, C, detect]. For more information on these pins, see annex section 1.d.

The main function to be used in the main application is RoboI2C.update\_devices(). It performs all hardware detection operations, updates all values read from the connected devices and returns a list of hardware detection events. Each event is in a dictionary form and parsing it using the keys provides all relevant information (MUX / channel ; connected / disconnected).

The dictionary lists ‘connected\_devices’ and ‘all\_devices’ hold all relevant device information. They are updated after a call to RoboI2C.update\_devices().

The I2C module requires device drivers that implement standard functionality. In order to detect new devices all drivers are tried out until one of them “works” (i.e. returns ‘True’ on getDeviceConnected method). More details on the drivers are provided below.

* I2C Drivers

The I2C drivers MUST implement the following methods:

* \_\_init\_\_(self, chn, ADR\_set = False): Class constructor must create the BeagleBoard mraa I2C object on the given channel with the known address. Most devices support an external address pin – if that is the case the driver can support this with the ADR\_set parameter.
* getDeviceConnected(self): Should return ‘True’ only if the device is identified and connected. Most devices have a ‘who am I’ register with a constant value that can be read. Reading this register and comparing the value is a good way of implementing this method. Also, mraa throws an exception if the read command is not acknowledged – thus any caught exception indicates that no device is connected.
* ConfigureDevice(self): Most devices do not work directly after a PowerOn reset. This function should configure the newly connected device to enter the operational state and start the measurement process. If there is any device that does not need configuring, leave this function blank, but DO NOT REMOVE IT – as it will trigger an error in the SoftWEAR I2C library.
* getAcceleractionXYZ(self): Currently, the SoftWEAR package is used to read accelerometer and IMU values over I2C. This function gets the 3 values from the connected device.
* getDevice(self): This function should return the device name as a string. This is required to properly inform the user about the connected device.
  1. ADC Module

All information presented in annex section 1.a are valid for the I2C module (including the translation dictionaries).

Also, the multiplexer configuration is identical to the one presented in annex section 1.b. Moreover, the same update mechanism as the one presented in annex section 1.b is implemented (SoftWEAR is consistent accorss it’s libraries).

The particularity of the ADC module lies in the channel connected hardware detection mechanism. All ADC channels require pull-down resistors. Thus, all the voltage readings should measure 0V when no device is connected. When a source of analog voltage is connected, the corresponding channel reading will go above 0V. An internal counter is implemented for debounce reasons – a number of RoboADC.timeout\_ticks consecutive readings are required to mark a channel as ‘active’ or ‘inactive’.

* 1. MUX Module

The MUX module is used to provide multiplexer features to other SoftWEAR libraries. The importing module should only create one class instance of the PWM module, as it encapsulates basic multiplexer functionality for multiple channels by using internal lists.

When adding a new multiplexer, 4 digital I/O pins are required:

* A, B, C: These pins are the MUX select pins. A corresponds to the least significant bit, while C is the most significant one. For example, 0b011 is A=1, B=1, C=0.
* detect: The detect pin should be pulled-down. When a MUX is connected, an electrical connection should be made between the detect pin and the VCC; thus pulling it to the ‘1’ logic level.

These pins are double-checked by using the Util Module (section 1.e) to avoid their usage in more than one place.

* 1. Util Module

This module currently only contains a list of used GPIO pins. This list is the same across all modules that import the Util package. It is used to keep track of used pins to avoid their concurrent usage. All other functionality that includes digital I/O pins should include and use this module.

Also, the list gpio2mraa is given. This gives all the available pure digital I/O that are not in the risk of being used by another peripheral. All SoftWEAR GPIO functionality should use this translation dictionary. Note that more pins can be added by editing the default BeagleBone configuration.

* 1. Com Module

The Communications module implements a data link using underlying TCP sockets. The general architecture of this module is presented in Figure 5.2 from section 5.1.

A background task is created that handles all the underlying socket blocking calls. Data is passed to and from the said task with the help of 2 internal queues.

When the function send\_data(self, data\_object) is called, the data object is first serialized then placed in the send queue. Serialization is performed in order to give the user the capability of sending dictionaries – thus simplifying the overlaying programs. On the next execution of the background task the send queue is emptied and all it’s contents sent over the network to the remote location.

When the function rcv\_data(self) is called, all objects present in the receive queue are popped and returned in a list. Objects are pushed in the receive queue in the background task; once they are deserialized.

An extra particularity of the Communications module is the remainder algorithm used in the receive process. Due to TCP incomplete transmissions, a remainder is used to hold incomplete data. The characters ‘{‘ and ‘}’ are used to detect complete data; i.e. when their counts are equal the data is considered complete. Because of this algorithm the sending and recieveing process is optimized only for dictionary objects which contain the curly braces.

* 1. Main Application

A main application demo is provided. It uses ADC, PWM and I2C with all their underlying features. Data is sent to the PC using dictionaries. Also, the PC sends PWM commands using the same dictionary based format.

Ideally, the main application is configured to run at start-up. Also, all messages that are sent to the PC could be sent via loopback address for use in other potential running programs, as presented in Figure 5.4 from section 5.4. The development of these embedded programs should greatly be simplified by the features offered by SoftWEAR.

## Known Issues

* 1. Connecting I2C devices are sometimes not properly configured.

Sometimes when continuously disconnecting and reconnecting I2C devices, they read out [0,0,0]. This is the value normally returned when they are not configured (but can be from other parts of the software). Re-connecting the device usually solves this. Maybe some delays have to be added in the device drivers.

* 1. At I2C MUX reconnect the BeagleBone sometimes crashes

After connecting 2 BNO055 and 1 MPU6050 devices to I2C channel 1 through a MUX, a MUX disconnection event occurred. When reconnecting the MUX with all attached devices, the BeagleBone once crashed (i.e. nothing was working on it anymore).

* 1. Having too many device drivers (with delays) can completley clog the BeagleBone

Since the only way of supporting plug-and-play detection with the current architecture is by scanning all channels (MUX included) with all drivers, scalability can become an issue. This is due to the amount of time spent executing the getDeviceConnected(self) function of the I2C device drivers. This family of functions executes (nb\_of\_channels \* nb\_of\_drivers) times at every sampling period. For example, if it takes ~1ms for completion and we have 8 channels with 10 device drivers, it may take up to 80ms to complete a hardware detection cycle. If this ever becomes an issue, ask Matteo about the solution with NV memory on I2C.

* 1. The default pin configurations have to be changed

## Hardware considerations

The presented SoftWEAR package only works on the following hardware configurations:

* 1. MUX hardware considerations

The schematic for connecting the MUX to the SoftWEAR platform is presented in Figure 5.3 from section 5.3. Note that, while not represented in the figure, the BeagleBone will provide power to the MUX. The detect pin is pulled down while the MUX (the black square) is disconnected. When the MUX is connnected, the power port (3V3) is connected to the detect pin pulling it high. This event is recognized in software. If the peripheral requires multiple multiplexers (e.g. I2C), they will share all pins except the ‘Common’ pins.

* 1. ADC hardware considerations

The schematic for connecting a new analog device to the BeagleBone is presented in Figure 5. Note that, while not represented in the figure, the BeagleBone will provide power to the analog device. The ADC pin is pulled down while the device (the black square) is disconnected. When the device is connnected, the ADC pin will no longer read 0.0V and the device connection is detected.

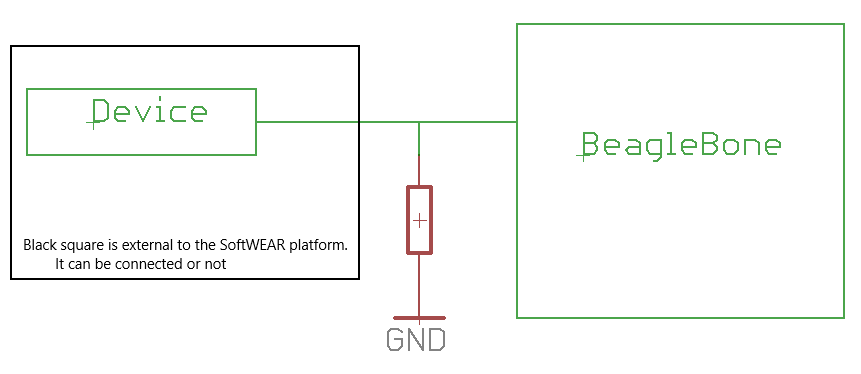


Figure 5. Analog hardware considerations. Device power and ground not represented.