SoftWEAR Software Design document

1. General Software Architecture and Software Unit interaction

The general architecture and the software unit interaction of SoftWEAR is presented in Figure 1 below.

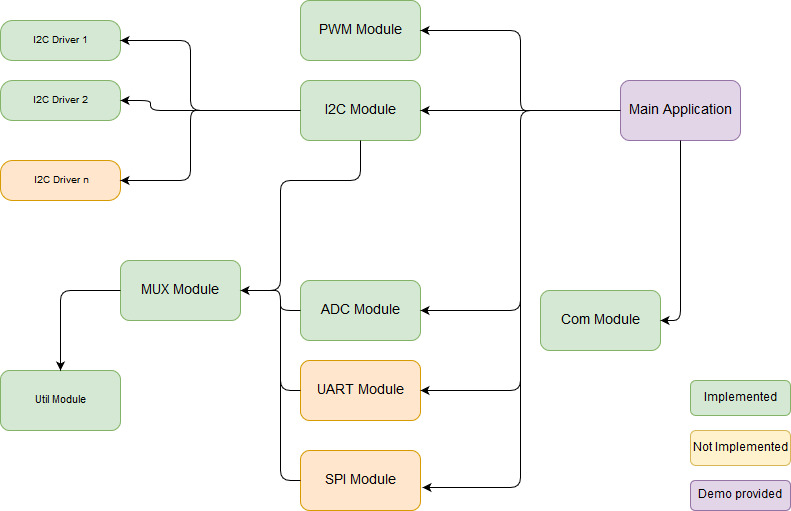


Fig. 1. SoftWEAR software architecture

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 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 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 section 1.a are valid for the I2C module (including the translation dictionaries).

Also, the multiplexer configuration is identical to the one presented in section 1.b. Moreover, the same update mechanism as the one presented in 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 (TODO).

* 1. Com Module

The Communications module implements a data link using underlying TCP sockets. The general architecture of this module is presented in Figure 2 below.

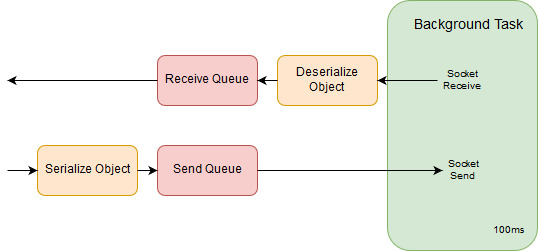


Figure 2. Communication Module Architecture

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 3 below. The development of these embedded programs should greatly be simplified by the features offered by SoftWEAR.

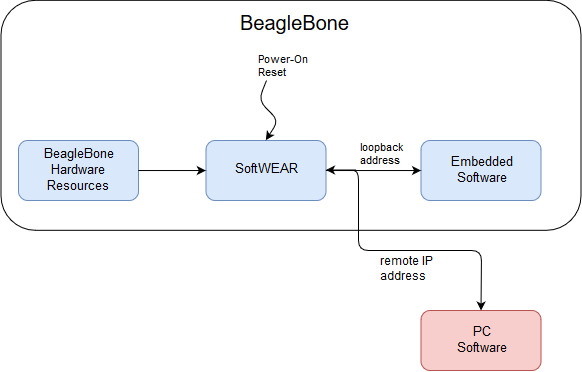


Figure 3. Adding Embedded Software applications running on top of SoftWEAR

1. 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 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 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

1. 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 4. 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.

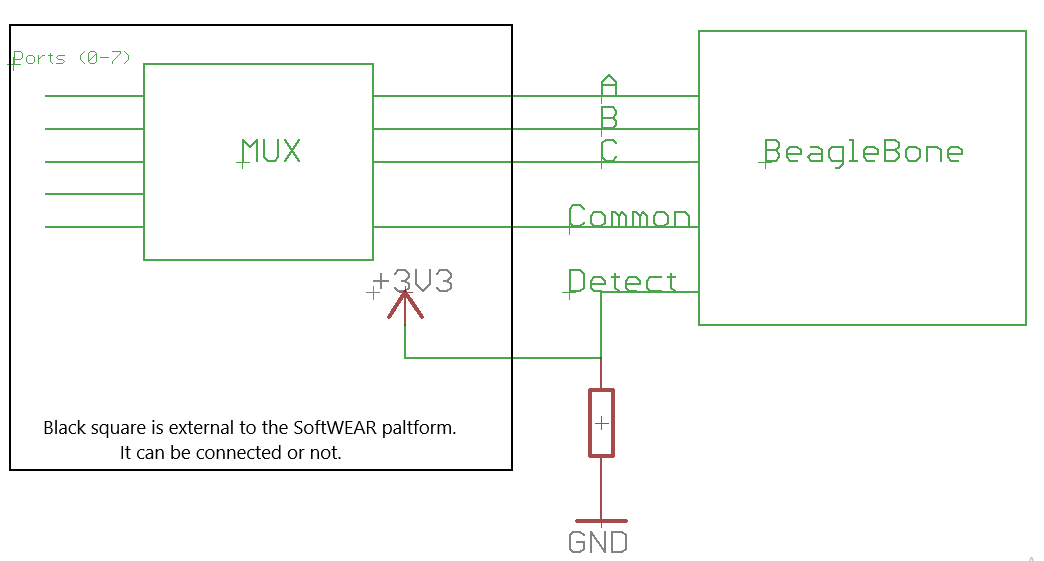


Fig. 4. Multiplexer hardware considerations. Multiplexer power and Ground not represented.

* 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.

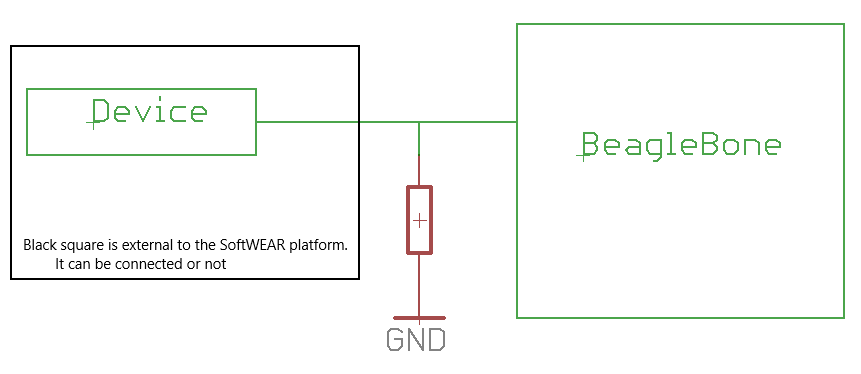


Figure 5. Analog hardware considerations. Device power and ground not represented.