Semester Thesis

Wireless sensors network for performance monitoring

Julien Romero

George Chatzipirpiridis

Adviser

Prof. Dr. Bradley J. Nelson

Institute of Robotics and Intelligent Systems

Swiss Federal Institute of Technology Zurich (ETH)

2016

To the world.

Preface

My work took place in the MSRL Lab in 2016. First, I would like to thank George Chatzipirpiridis and Alessandro Schr for their precious advice which made the project possible. Then, I would like to thank all other people in the lab who helped me by introducing me to the workshop or by giving me good ideas. I hope my work will be useful to others after me.

Abstract

Sensors are getting more and more popular nowadays for performance monitoring due to the high interest in Big Data and Internet of Things. There are now use even outside industry, in smart home or smart cities for instance. We explore here ways to build a wireless sensors network in a flexible and scalable way.

Zusammenfassung

Kurzfassung der Arbeit.

Contents

List of Figures

1 Introduction

We want to build a wireless sensors network for performance analysis. The final goal is to have different kinds of sensors connected together and to be able to visualize measurements and to configure them from a GUI.

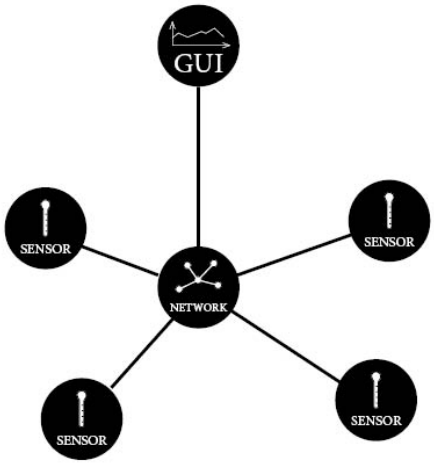

Figure 1: General architecture.

To begin with, wireless communication protocols are explored. Once one is chosen and a hardware implementation is picked, we start exploring the system and show that drivers for peripherals can easily be written. Next, we continue with the interesting part of the system, the communications, and we finish by a method to share a common time among all nodes of the network.

All our work is available on our github : `https://github.com/Aunsiels/Mesh_Bee`

2 Choosing a Wireless Protocol

The first thing to think about is how to communicate. There are plenty of available protocols and we have to choose among them the one which will best fit to our needs. Eight protocols are considered: ZigBee, Bluetooth, BLE, Rubee, Zwave, ANT/ANT+, EnOcean and WiFi. To evaluate them, four characteristics are chosen: the data rate, the size, the consumption and the range. Of course, this properties depends of a lot of parameters : hardware, environment,... So, we try to choose values which appeared in most of the descriptions. The use cases are also useful to know what kind of things are possible with a protocol.

2.1 The Protocols

2.1.1 Zigbee

Zigbee is based on 802.15.4 specification, at 2.6 GHz. It is used in smart home, industrial control, medical tools, fire sensors... It is also used for low energy sensors, which is useful for us. A Ad Hoc network is created around a coordinator. Different architecture of the network are possible, we shall see it later. However, there can be problems with inferences as 2.4GHz is often used.

2.1.2 Bluetooth

Bluetooth is based on IEEE 802.15.1 and works at 2.6GHz. It is used in smart phones, keyboards, mice, sensors,... The fact that it is in all smart phone popularized in. The consumption is medium.

2.1.3 BLE

The Bluetooth Low Energy, just like Bluetooth, works at 2.4GHz. We can see it as a low consumption version of Bluetooth, which exists to answer modern problems like consumption. So, it is used in health care, sport, sensors...

2.1.4 Rubee

Rubee is a 450kHz protocol designed to transmit information in harsh environment (with a lot of steel for example), with high security. So, it is often used for

military applications. It works with tags, a bit like NFC but the range can be higher. However, it is not that easy that find chip for Rubee development.

2.1.5 Zwave

Zwave is a protocol around 900MHz, designed for home automation applications. It can be integrated to a lot of existing of house products. It has a small consumption and a lower data rate than Zigbee.

2.1.6 ANT/ANT+

ANT/ANT+ works on 2.4GHz. It is a very low consumption protocol. It is mainly used in health, sport, smart home and industry.

2.1.7 EnOcean

EnOcean works on 902 MHz, 928.35 MHz, 868.3 MHz and 315 MHz and is a very low energy protocol (can also work without battery, inside a switch for example). The data rate is low. It can be used in smart house and in sensors.

2.1.8 WiFi

WiFi is a very popular protocol in personal application and is based on IEEE 802.11 . It operates on 2.4, 3.6, 5, and 60 GHz frequency bands. The speeds that can be achieve are high but the consumptions are also high. It is also used for the Internet of things and a lot of ship are now design with wifi integrated (Raspberry Pi 3, Arduino Genuino MKR1000, Red Pitaya, Electric Imp...).

2.2 Comparing Protocols

Everything is summarized in an odt file available on the github. The results are on sheet 1 and 2 and some results were plotted.

We can see that each protocol is in a different area and so, we need to choose the one which will be good for us. For that, a cost function was design according to our needs. The function has four parameters (for the four protocol characteristics) that can be tuned. The chips available are also considered. Are they hard to get ? How easy is it to program them ? Is their help for it somewhere ? What are the prices ? Is it open sourced ? Finally we decide to go for Zigbee.

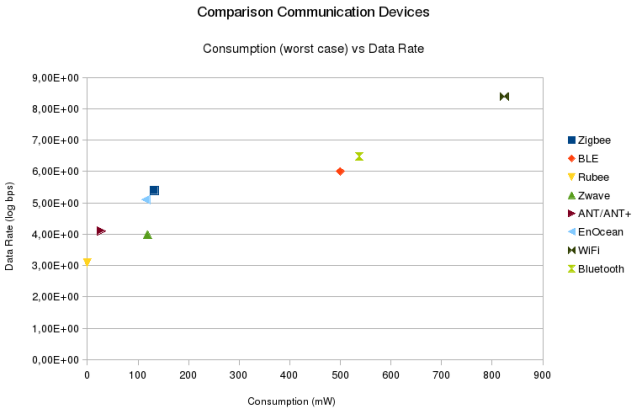

Figure 2: Consumption vs Data Rate

2.3 Choosing a Zigbee Chip

There are plenty of Zigbee chips available. What we want to find is a chip which controls both Zigbee and a sensor. So, the prototype can be programmed on the chip directly, without having to use an additional board like an Arduino. We also need to be able to read sensors thanks to I2C and ADC. There is a nice design, based on a NXP chip : the MeshBee. It is open source (both software and hardware) and easy to prototype with. Thus, thanks to it, we are able to first try to write a program to test if the prototype is working and then we will design a new chip in which sensors are directly integrated. That makes the final result smaller.

3 Programming the Meshbee

The embedded software on the MeshBee have to read data from a sensor and to send information through the network to the main node, the coordinator.

3.1 Meshbee Framework

First things first, we need to understand how MeshBee works.

The MeshBee firmware is built on an OS provided by NXP, JenOs, and a SDK for Low Energy Zigbee, also provided by NXP. It is important to know that because sometimes, the functions we are looking for (as interrupt-liked functions)

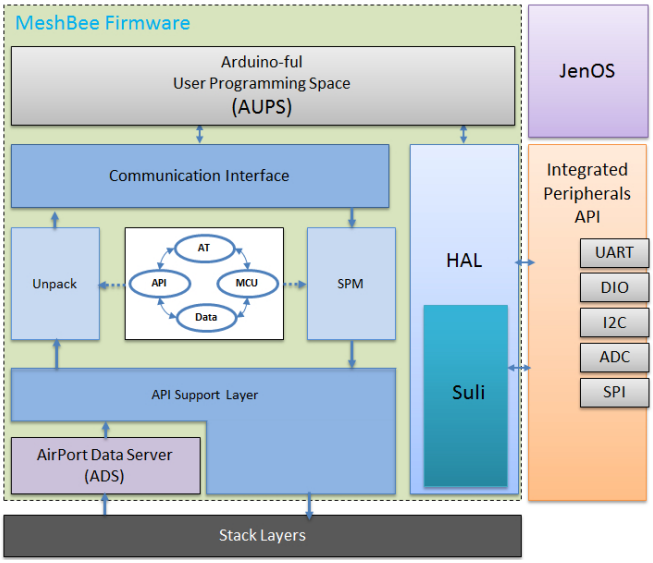

Figure 3: MeshBee Architecture

are not in the MeshBee Framework but directly in the library provided by NXP.

Inside the MeshBee, we find an Arduino programming environment. The AUPS (Arduino-ful User Programming Space) provide two functions : a setup and a loop function. The setup function is called during the initialization of the system and the loop function is called periodically (it is possible to specify the period).

In the middle of Figure 3, there are the four modes of the MeshBee :

- AT : this is an interactive mode. Easy commands have to be send to the MeshBee and it answers in an user-friendly way. To go to AT mode from everywhere, +++ have to be send.
- API : this is a more effective and formatted way to communicate with the MCU. Thanks to it, AT function can be called from inside and outside the MCU, via UART for example. It is also possible send query to other nodes through the network thanks to the API commands.
- MCU : this is the Arduino mode. The Arduino loop is only executed if the MeshBee is in this mode.
- Data : this is a transparent mode : all data received on the Zigbee network are directly transmitted to UART and all data sent via UART are broadcast

on the network.

The last thing to understand about this firmware is the Suli interface. This is a general library created by SeeedStudio. The goal is to make the interaction with GPIOs easier. So, for I2C, ADC,... there is no need to call the functions from JenOs but the easy functions provided by Suli can be used. However, we will need more advanced functions.

3.2 Meshbee Hardware

Figure 4 presents a picture of the hardware with the pin layout and Figure 5 shows the pin layout in more details.

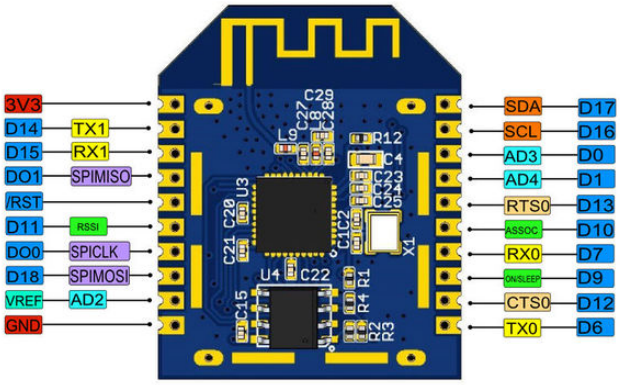

Figure 4: MeshBee Hardware

Pin	Alternate Functions	Type	Description
3V3	-	Supply input	VCC, +3.3V
TX1	D14	Output	Uart1 Tx port; Digital IO 14
RX1	D15	Input	Uart Rx port; Digital IO 15
DO1	SPIMISO	Input/Output	Digital Output 1; SPI Master In Slave Out Input
!RST	-	Input	Reset port
D11	PWM1	Input/Output	Digital IO 11 (default usage: RSSI Indicator); PWM1 Output
DO0	SPICLK	Output	Digital Output 0; SPI Master Clock Output
D18	SPIMOSI	Input/Output	Digital IO 18; SPI Master Out Slave In Output
VRef	ADC2	Input	Analogue peripheral reference voltage; ADC input 2
GND	-	GND	GND
TX0	D6	Input/Output	Uart0 Tx port; Digital IO 6
D12	CTS0	Input/Output	Digital IO 12; UART 0 Clear To Send Input
D9	-	Input/Output	Digital IO 9 (default usage: Mesh Bee ON/Sleep Indicator)
RX0	D7	Input/Output	Uart0 Rx port; Digital IO 7
D10	-	Input/Output	Digital IO 10 (default usage: Network Association Indicator)
D13	RTS0	Input/Output	Digital IO 13; UART 0 Request To Send Output
D1	SPISEL2; ADC4	Input/Output	Digital IO 1; SPI Master Select Output 2; ADC input 4
D0	SPISEL1; ADC3	Input/Output	Digital IO 0; SPI Master Select Output 1; ADC input 3
D16	SCL	Input/Output	Digital IO 16; I2C clock
D17	SDA	Input/Output	Digital IO 17; I2C data

Figure 5: MeshBee Pin Layout

3.3 Tools

NXP provides tools to program the Meshbee. These tools include a programmer and libraries to code the JN516X. More details about the installation are provided on our github : `https://github.com/Aunsiels/Mesh_Bee/blob/master/doc/documentation.md`.

3.4 AT mode

We have everything to compile so we are ready to communicate with the MeshBee. This communication uses **UART1**, so any USB-to-TTL working at **3.3V** (this is very important) device would work. For now, we only use the official programmer. We just switch the left switch to UART1 (up). Then, a serial communication tool is used to communicate. The Baud is **115200**, there is **no parity bit**, **8 data bits**, **1 stop bit** and **no flow control**.

When we send an AT command, it is important to have **CR** (which is $\mathring{)}$ at the end of the line. It will not be specify in the rest of the text.

It is time for our first command : +++. This allows us to go to AT mode. We receive an **Enter AT Mode** if we were not in AT mode yet or an **Error, invalid command** if we were in AT mode.

AT commands are composed as follow : **ATXX[YYYY]**. They always begin by AT. Then follow two characters to identify the command, for example IF or AJ. Then follow up to 4 optional numbers which are parameters of the command. For more about it, everything was explained in details in the AT commands section in the User's Manual. We note by looking at the code (that's something we have to often do) that all commands are not available for all device type. For instance, an end-node is not able to create a network and thus do not have this command.

Here is a list of useful commands. We begin with **ATIF**. This prints information about the MeshBee. We check that the Device Type is the one we think, the coordinator for example. With the coordinator, we send **ATPA1**. This command creates a new network. The MeshBee is reset and the ASSOC led on the programmer light up. We send **ATIF** again and read the **PANID** field, which is the id of our network. We can go to an other mode, for example data mode with **ATDT**.

Next, an other node is connected to the network. We have to remember that

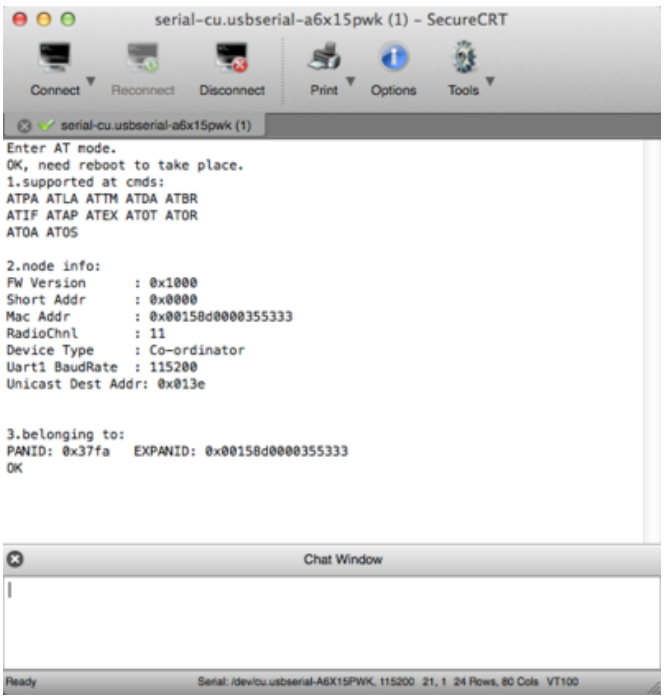

Figure 6: Result ATIF

we always need the coordinator to be on. We unplug it from the programmer and give it power with the raspberry pi (see below). We want to be sure that it is connected, so a LED is also connected (and a resistor !) to the ASSOC pin (We were able to read the names of the pins below the MeshBee but there is also Figure 4 above to check). A node is plugged in the programmer. It had been flashed with the router or end-node code.

As before, we go to AT mode with +++ and get information with ATIF. If the MeshBee was not configured before, the PANID is 0x0000 or a number different from the one of the coordinator. We have to say to our device to automatically join a network with **ATAJ1**. Then the network is rescanned with **ATRS** and after some seconds, the ASSOC led light up. We check information with ATIF and then we have the same PANID than the coordinator. To know what are the other nodes on the network, we send the **ATLA** command. That sends a request to all nodes and we receive information about all of them. We go to data mode with **ATDT** and everything which is typed in the console is broadcast on the network. As the coordinator is also linked to a serial port, the raspberry pi one (see below), we see that our messages appear in the coordinator console.

3.5 MCU mode

The MCU mode is the "Arduino like" mode. We go to this mode from the AT mode by just typing **ATMC**. Then, the Arduino-loop start.

3.5.1 The Arduino-like functions

The source code is located in **src**, on the github. One of the file is called ups_arduino_sketch.c . This file contains the two Arduino-like functions : **arduino_setup** and **arduino_loop**.

The **arduino_setup** function is called when entering in MCU mode, through the function ups_init we find in firmware_aups.c (AUPS stands for Arduino User Programming Space). This function also calls suli_init, so there is no need to initialize Suli (see below) if the device is in MCU mode. In the arduino_setup function, everything that needs to be done at the beginning of our program is written, for example initialization of peripherals.

The **arduino_loop** was a function which was called periodically. We define the time between two loops by typing in AT mode, **ATMFXXXX** where XXXX is a number between 0 and 3000 and is the time in milliseconds between two Arduino_Loop. This Arduino_Loop function is a task, defined in firmware_aups.c, and either calls arduino_loop or exits MCU mode to AT mode if +++ is read.

We notice that here we have to use the C preprocessor to know for which device we are compiling the program for.

```
#ifdef TARGET_COO
// Code for the coordinator
#elif TARGET_ROU
// Code for the router
#else
// Code for the end node
#endif
```

3.6 The SULI Library

SULI is a library written by SeeedStudio to easily use the peripherals. It is imported it by just adding :

```
#include "suli.h"
```

When the device is in MCU mode, we do not need to initialize Suli. Otherwise :

```
suli_init();
```

3.6.1 Simple GPIO

Our first experiment is with simple input/output. First, we need to initialize the pin. We need to use the **void suli_pin_init(IO_T *pio, PIN_T pin**) function. It takes two arguments : an IO_T variable that contains the properties of the pin and the pin number.

```
IO_T led_io;
suli_pin_init(&led_io, D9);
```

The pin number is either an int or a defined pin name (they are defined in suli/suli.h) :

```
D0 = 0, D1=1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11,
D12, D13, D14, D15, D16, D17, D18, D19, D20, DO0=33,
DO1=34, A3=0, A4=1, A2=50, A1, TEMP, VOL
```

Then we have to set the direction, input or output, of the pin. We use the **void suli_pin_dir(IO_T *pio, DIR_T dir)** function, which takes a IO_T argument and a direction, which could either be **HAL_PIN_OUTPUT** or **HAL_PIN_INPUT**.

The state of a pin can be read with **int16 suli_pin_read(IO_T *pio)** and be written with **void suli_pin_write(IO_T *pio, int16 state)**. The state can either be **HAL_PIN_HIGH** or **HAL_PIN_LOW**.

A pulse could be read with **uint32 suli_pulse_in(IO_T *pio, uint8 state, uint32 timeout)**, which returns the length of the pulse in microseconds.

3.6.2 ADC

Like for simple input/output, the pin need to be initialized for analog reading first. We use **void suli_analog_init(ANALOG_T * aio, PIN_T pin)** where ANALOG_T was the equivalent of IO_T. For the pin, there is a choice between

A3=0, A4=1, A2=50, A1, TEMP, VOL. Then, we are able to read with **int16 suli_analog_read(ANALOG_T *aio)**.

```
ANALOG_T temp_pin;
suli_analog_init(&temp_pin, TEMP);
int16 temper = suli_analog_read(&temp_pin);
```

3.6.3 Time functions

We find two functions in Suli to wait a certain amount of time : **void suli_delay_us(uint32 us);** for microseconds waiting and **void suli_delay_ms(uint32 ms);** for milliseconds waiting. We also find a function to know for how long the program have been running : **uint32 suli_millis(void);** for milliseconds and **uint32 suli_micros(void);** for microseconds. There could be overflow, after 50 days for suli_millis but after 70 minutes for suli_micros.

Because of these overflows, the time functions are modified to return an uint64. This will be important for time synchronization.

3.6.4 I2C

I2C works the same way than ADC and input/output. There is an initialization function **void suli_i2c_init(void * i2c_device);**, a write function **uint8 suli_i2c_write(void * i2c_device, uint8 dev_addr, uint8 *data, uint8 len);** and a read function **uint8 suli_i2c_read(void * i2c_device, uint8 dev_addr, uint8 *buff, uint8 len);**. For I2C, as we had no choice for the pin, specify the i2c_device is useless. We just gave NULL.

```
suli_i2c_init(NULL);
uint8 data = TRIGGER_HUMD_MEASURE_NOHOLD;
suli_i2c_write(NULL, HTDU21D_ADDRESS, &data, 1);
uint8 msb;
suli_i2c_read(NULL, HTDU21D_ADDRESS, &msb, 1);
```

We write a use example of I2C in **src/humidity.c**. This is a driver to read data from a humidity and temperature sensor (see below).

3.6.5 UART

Like I2C, we have no choice for the UART Port, it has to be UART1. So, in the initialization function **void suli_uart_init(void * uart_device, int16 uart_num, uint32 baud);**, there is no need to specify uart_device and uart_num. The baud can be : 4800, 9600, 19200, 38400, 57600 or 115200. We notice that by default, the baud rate is initialized to 115200, so there is no special need to initialize it.

For the writing part, there is a general function, **void suli_uart_send(void * uart_device, int16 uart_num, uint8 *data, uint16 len);** which just takes an array of data and the length of this array. Then, there are more specific functions : **void suli_uart_send_byte(void * uart_device, int16 uart_num, uint8 data);**, **void suli_uart_write_float(void uart_device, int16 uart_num, float data, uint8 prec);**, **void suli_uart_write_int(void * uart_device, int16 uart_num, int32 num);**. However, we want to use printf-like function, much easier to use. So, most of the time, we used **void suli_uart_printf(void uart_device, int16 uart_num, const char *fmt, ...);** (the three dots are a notation of C to say that the number of argument is undetermined, as it is in printf). This function does not work well for float.

```
suli_uart_printf(NULL, NULL, "<HeartBeat%d>\r\n",
                        random());
```

For the reading part, we have a function to know if there is something to read : **uint16 suli_uart_readable(void *uart_device, int16 uart_num);** which returns one if UART has received readable data, zero otherwise. Then we are able to read a byte with **uint8 suli_uart_read_byte(void *uart_device, int16 uart_num);**

3.7 API MODE

The API is a simple way to communicate with the MCU from outside. The messages have a particular form, which is explained in details in the User's Manual. It is also possible to call API commands from MCU mode.

3.7.1 API Frames

The API frame must have a specific form, which depends on what we want to do. The frames are described in the User's Manual.

3.7.2 Send API Commands from MCU Mode

To send an API Command, we need to build a tsApiSpec, which is a representation of an API Frame. Here is what is inside :

```
/* API-specific structure */
typedef struct
{
    uint8 startDelimiter;
    // start delimiter '0x7e'
    uint8 length;
    // length = sizeof(payload)
    uint8 teApiIdentifier;
    //indicate what type of packets this is
    union
    {
        /*diff app frame*/
        uint8 dummyByte;
        //dummy byte for non-information frame
        tsNwkTopoReq nwkTopoReq;
        tsNwkTopoResp nwkTopoResp;
        tsLocalAtReq localAtReq;
        tsLocalAtResp localAtResp;
        tsRemoteAtReq remoteAtReq;
        tsRemoteAtResp remoteAtResp;
        tsTxDataPacket txDataPacket;
        tsOtaNotice    otaNotice;
        //OTA notice message
        tsOtaReq otaReq;
        tsOtaResp otaResp;
        tsOtaStatusResp otaStatusResp;
```

```
    }__attribute__ ((packed)) payload;
    uint8 checkSum;       //verify byte
}__attribute__ ((packed)) tsApiSpec;
```

We do not care about startDelimiter, it was always 0x7e and is set for us by a function. Then come the length of the payload, which is the useful data in the frame. The teApiIdentifier is the id to identify a packet. It can be one of the following option :

```
typedef enum
{
    /* API identifier */
    API_LOCAL_AT_REQ = 0x08,
    //local At require
    API_LOCAL_AT_RESP = 0x88,
    //local At response
    API_REMOTE_AT_REQ = 0x17,
    //remote At require
    API_REMOTE_AT_RESP = 0x97,
    //remote At response
    API_DATA_PACKET = 0x02,
    //indicate that's a data
    //packet,data packet is certainly remote packet.
    API_TEST = 0x8f,
    //Test
    API_OTA_NTC = 0xd3,
    API_OTA_REQ = 0xb0,
    API_OTA_RESP = 0x06,
    API_OTA_ABT_REQ = 0xf7,
    API_OTA_ABT_RESP = 0xdb,
    API_OTA_UPG_REQ = 0x5a,
    API_OTA_UPG_RESP = 0xe6,
    API_OTA_ST_REQ = 0x91,
    API_OTA_ST_RESP = 0x89,
    API_TOPO_REQ = 0xfb,
    API_TOPO_RESP = 0x6b
```

```
}teApiIdentifier;
```

Depending of what we are sending, we have to choose the correct id. They are described in the User's Manual. Then, there is a union, payload. In C, it simply means that we have to choose one of the following options. The __attribute__((packed)) means that the compiler has to put all the fields together, without a hole, in the memory : they are "packed"". We notice that in the payload, we have for example localAtReq to make, as the name said, a local At request. To find the exact definition of a payload, we go to **include/-firmware_at_api.h**. Then, finally, we have a checksum which helps us to be a more sure that data are transmitted without problems (like a bit switch).

Most of the time, we do not have to complete the tsApiSpec ourself. Functions are provided to make it easier. It is also the case for tsLocalAtResp and tsRemoteAtResp.

For example, if we want to send data to another node in the network. It could be done by sending a API command.

```
#include "firmware_at_api.h"
#include "firmware_api_pack"
//Those two include contains the function
//to easily create tsApiSpec and send them

tsApiSpec apiSpec;
uint8 tmp[sizeof(tsApiSpec)]={0};
// Will contain the final result
//For now we put in it a string
sprintf(tmp, "TEST %d", 3);
// Just a formatted string (like in printf) in tmp
PCK_vApiSpecDataFrame(&apiSpec, 0xec,
                      0x00, tmp, strlen(tmp));
// This function create a DataFrame
uint16 size = i32CopyApiSpec(&apiSpec, tmp);
//We actually create the array containing
//the API Frame in tmp
API_bSendToAirPort(UNICAST, 0x0000, tmp, size);
// We send the message to 0x0000, the coordinator
```

To fill tsApiSpec, we use, for data, **void PCK_vApiSpecDataFrame(tsApiSpec *apiSpec, uint8 frameId, uint8 option, void *data, int len);** in firmware_api_pack.h. The frameId just identified the frame, we put whatever we want, it is not that important. The option is 0 for UNICAST, 1 for BROADCAST. It is also important to create the actual frame inside an array with **int i32CopyApiSpec(tsApiSpec *spec, uint8 *dst);** also in firmware_api_pack.h. It returns the size of the frame, which is useful when we want to send my frame with **bool API_bSendToAirPort(uint16 txMode, uint16 unicastDest, uint8 *buf, int len);** in firmware_at_api.h. The txMode can be either UNICAST (to only one node) or BROADCAST (to all nodes). Then we have to specify an address which is only used if we are in UNICAST mode (otherwise it does not matter), the frame array we have just created and its length.

We are also able to call AT commands thanks to **int API_i32AtCmdProc(uint8 *buf, int len);** in firmware_at_api.h. We just give it a command and its length and it returns if the function has succeeded or not.

```
char * aj = "ATAJ1";
API_i32AtCmdProc(aj, 5);
```

We find other functions in firmware_at_api.h and firmware_api_pack.h but for now we do not find them useful. It is for example possible to send a message to a given Mac address. We explore the code if we want to go in more details.

3.8 Create New Tasks

Sometimes, we may need to call a given function every one second. In JenOs, we need to create a new **Task**. Tasks are special functions which can only be called by the OS. They are run in parallel of the main function (so we need to be careful with shared resources). The first thing we have to do is to say to the OS to add a new task. This is done in the file **src/MeshBee.oscfgdiag**. When we open the raw files, there is a lot of text and it is hard to understand everything. NXP provides a graphical interface to edit this file. We use it.

In the Jennic/Tools, we can find eclipse, a well-known code editor. The file is still a raw text when we open it. We have to choose plugins in Jennic/Tools/Eclipse_plugins. Once the plugins are installed, the raw text becomes a diagram.

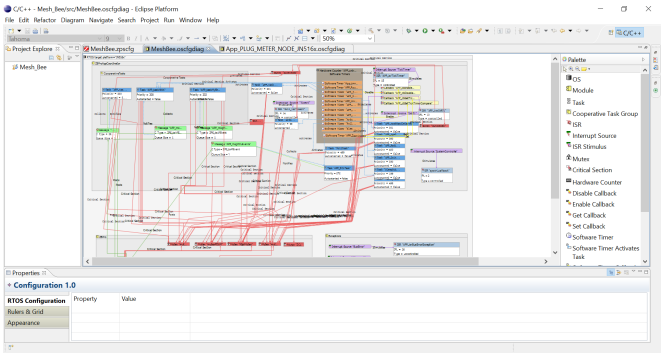

Figure 7: Eclipse configuration file

We see, on the left, the project explorer where are our files. On the right, there are the components that can be added in the diagram, which is in the middle if it was opened. At the bottom, there are the properties of the selected component. If it is not there, it can be added by clicking on the bottom left hand corner icon and by selecting "Properties".

We look at diagram in more details.

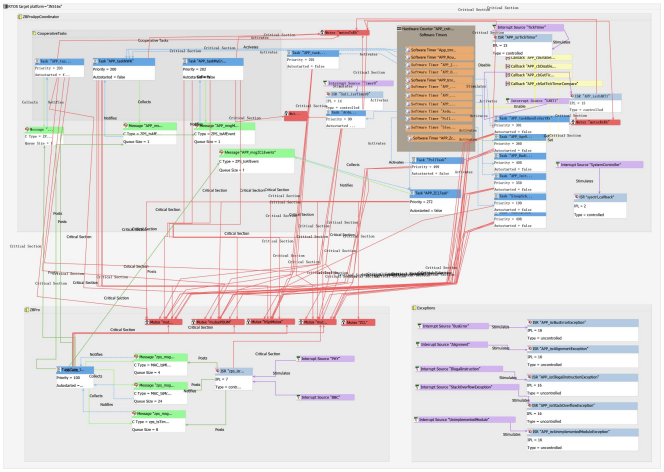

Figure 8: Details configuration

We can see that there are different kinds of blocks : Tasks, Software Timer, mutexes... These blocks are linked to each other in a meaningful way. For instance, a task is able to use a mutex to enter in a critical section. We notice that sometimes the documentation of JenOs assumes that some blocks are declared. For example, it is assumed that there is a task declared for time synchronization or that there is an interrupt handler for System Controller's interrupts. So, the

first time we discovered it, it was a bit surprising that given function do not work. If it was written to give a callback function for an interrupt, even if the function existed, if there was nothing in the configuration file to declare the interrupt and link it to an ISR (see later to know what it is), the function was useless and did not work at all.

We create a task to synchronize time. The task name was APP_ZCLTask. The goal of the task is to keep the count of the time (see later). We just select Task in the Palette. Then, we click in a free space in the diagram. The Task box appears. We give it a name and a priority (for the scheduler). Then are created links with other components. Some links can be created directly by using the input and output arrows at the border of the box (they appear when passing the mouse above the box). Some links need to be selected in the Palette. It is the case for Critical Section.

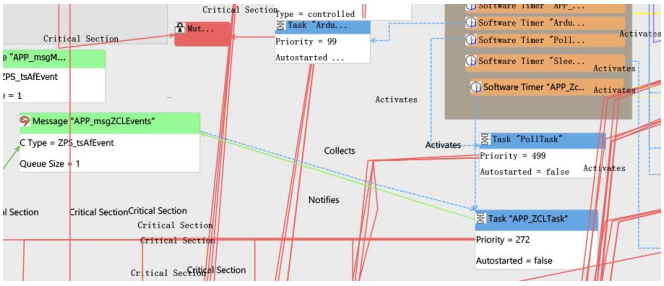

Figure 9: Zoom configuration

The task is declared to the OS. A file will be generated at compile time to transform those information into code. The files generated are called build/gen/os ×
.

We then have to add code to explain to the OS what to do with the task. To declare the task, we use the macro : **OS_TASK(task_name)**. For example, if the task's name was App_Task, it looks like :

```
OS(APP_Task) {
    // Code
}
```

The task is created. We need to say when to execute our task. The easiest way to it is to use the function **OS_teStatus OS_eActivateTask(OS_thTask hTask);**. It takes the task name as an argument and returns if it succeeded

or not : OS_E_OK (successful), OS_E_BADTASK (invalid task handle used) or OS_E_OVERACTIVATION (maximum number of activations exceeded: 65535). It asks the OS to schedule the given task. However, it is quite limited because we need to know exactly when to call the task. However, we need to call the task every second. So, we need to use a timer, the one we declared in our configuration file. We have a function to say to start the timer : **OS_teStatus OS_eStartSWTimer(OS_thSWTimer hSWTimer, uint32 u32Ticks, void *pvData);**. It takes the timer's name, the number of ticks before the timer ends (we used the macro **APP_TIME_MS(n_milliseconds))**, and data which are only useful for callback functions (here we provided NULL for our task). It returns a status which could be OS_E_OK (successful), OS_E_BADSWTIMER (invalid software timer handle) or OS_E_SWTIMER_RUNNING (software timer already running). Once our task is called, it is either possible to restart the timer with the same function or to continue to use the same timer (which continues to count while we do some computation) with **OS_teStatus OS_eContinueSWTimer(OS_thSWTimer hSWTimer, uint32 u32Ticks, void *pvData);** (returns the same status than OS_eStartSWTimer). Finally, we need to check the status of the timer with **OS_teStatus OS_eGetSWTimerStatus(OS_thSWTimer hSWTimer);** to know whether it has just expired or not. It returns either OS_E_BADSWTIMER (invalid software timer handle), OS_E_SWTIMER_RUNNING (software timer is running), OS_E_SWTIMER_STOPPED (software timer has been stopped) or OS_E_SWTIMER_EXPIRED (software timer has expired).

For applications, see later, in time synchronization v1 for example.

3.9 Interrupt From a Button

To try our time synchronization, we need a physical common event. Our first experiment is carried out with a simple button. Later, it will be replaced by an interrupt from a sensor for example, to trigger a measurement for instance. An interrupt is something which has to be scheduled by the OS, so we need to declare it in the configuration file. The document claims that they have a function a register a callback for I/O interrupts but it does not work. So, we declare an interrupt source, System Controller. System controller gathered different kind of interrupt among which the IO ones. Then we declare a task, sysctrl_callback, which was stimulated by the interrupt source we have just created.

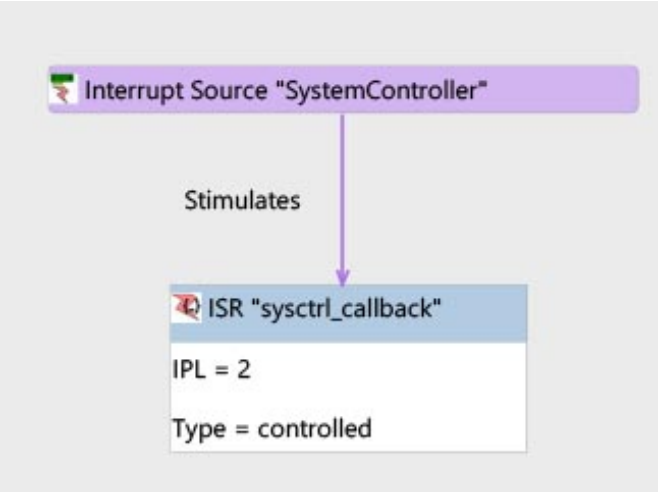

Figure 10: Zoom Interrupt

The interrupt function is defined in **src/utils_meshbee.c**. It sends a simple frame to the server when the button is pressed. The button is initialized in the arduino_setup function. Suli was not enough so we have to use functions from JenOs :

```
uint32 mask = (1 << D0);
vAHI_DioSetDirection(mask, 0);
vAHI_DioInterruptEdge(0, mask);
// First argument for rising edge
vAHI_DioInterruptEnable(mask, 0);
```

The mask simply indicats which IOs are concerned by a change. We need to set to 1 the bit at the position of the number of the IO. **vAHI_DioSetDirection** set the direction of the pin, the first arguments are the pins we want to change to input and the second to output. **vAHI_DioInterruptEdge** allowed us to change the edge which triggers the interrupt, the first argument being falling and the second rising. Then **vAHI_DioInterruptEnable** was used to enable the interrupt (first argument) or to disable it (second argument).

4 Examples

The first experiments we carry out allow us to better understand how it works. We begin with a simple hello word, with a LED. We will continue with more

complex peripherals.

4.1 Hello World !

The circuit is really simple : a Led and a resistor connected to D9.

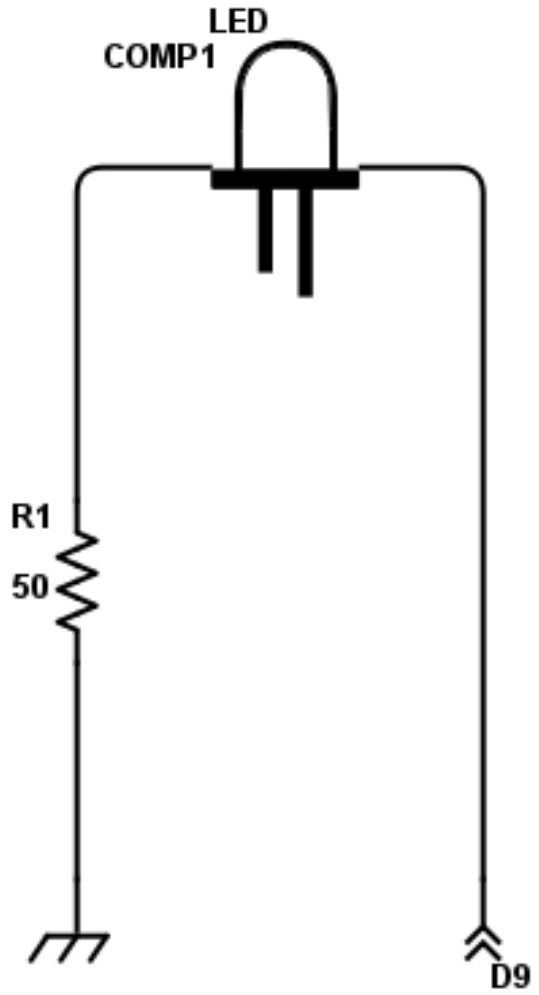

Figure 11: Simple Led

The code we use is also simple.

```
#include "suli.h"
```

```
IO_T led_io;
int state;

void arduino_setup(void){
    suli_pin_init(&led_io, D9);
    suli_pin_dir(&led_io, HAL_PIN_OUTPUT);
    state = HAL_PIN_OUTPUT;
    // contains the current state of the LED
    suli_pin_write(&led_io, state);
}

void arduino_loop(void){
    state = ~state;
    // We just exchange the state of the pin

    suli_pin_write(&led_io, state); //Blink

    suli_delay_ms(1000); // wait one second
}
```

Note that instead of waiting actively with suli_delay_ms, we could also change the period of the arduino loop with ATMF1000.

4.2 Read a Brightness

To measure the brightness, we use a photo-resistor : the higher the brightness the lower the resistance. So, to measure it, we do a voltage divider with a resistor of 5K ohms.

We connect the middle point with ADC3, i.e. D0. Here is the code, we read the brightness and send it to the coordinator :

```
#include "suli.h"
#include "firmware_api_pack.h"
#include "firmware_at_api.h"

ANALOG_T brightness_pin;
```

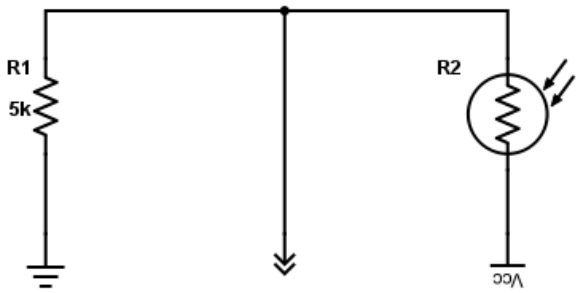

Figure 12: Photoresistor

```
void arduino_setup(void){
    suli_analog_init(&brightness_pin);
}

void arduino_loop(void){
    uint8 tmp[sizeof(tsApiSpec)]={0};
    tsApiSpec apiSpec;

    int16 bright = suli_analog_read(brightness_pin);
    sprintf(tmp, "BRIGHTNESS%ld\r\n", bright);
    PCK_vApiSpecDataFrame(&apiSpec, 0xec, 0x00, tmp,
                          strlen(tmp));

    uint16 size = i32CopyApiSpec(&apiSpec, tmp);
    if(API_bSendToAirPort(UNICAST, 0x0000, tmp, size))
    {
        suli_uart_printf(NULL, NULL,
            "Brightness read %ld", bright);
        }

    suli_delay_ms(1000); // wait one second
}
```

Note that we could also read the internal temperature the same way : the initialization of the pin was replaced by **suli_analog_init(&temp_pin, TEMP);**.

4.3 A Driver for HTU21D

HTU21D is a humidity sensor which can also output a temperature. It is communicating by I2C with the MCU. The code for it is in **src/humidity.c**. There are three functions. The first one is **void init_humidity(void)**. It simply initializes everything to read the sensor, here it is only initializing I2C.

Then we write two functions. One to read humidity : **unsigned int read_humidity(void)**. It returns a raw humidity. To get the real one, we just computed :

$-6 + (125 * rawHumidity/(float)65536)$;

The function requests to read humidity to HTU21D. Then, it waits for the result to be available and read it. The message read is composed of three parts : the most and least signification bytes of the humidity and a checksum to be sure of what we read. We note that inside the humidity value, there is two status bits we need to erase.

The temperature reading works the same way, with **unsigned int read_temperature(void)**. It returns the raw temperature value, so to get the real one, we need to compute :

$(float)(-46.85 + (175.72 * rawTemperature/(float)65536))$

4.4 A Driver for LSM9DS0

We would like to detect taps. To do so, we need an accelerometer which can generate interrupts on taps. We have a LSM9DS0, so we use it. To communicate, we use I2C. In the meantime, there is a gyroscope and a magnetometer, so we use it. The driver could be found in **src/LSM9DS0.c**. To initialize the LSM9DS0, we write an initialization function, **uint16 init_LSM(LSM_parameters params);**. It takes the parameters to configure the LSM9DS0. They are defined in **include/LSM9DS0.h**. Then, we calibrate the gyroscope and the accelerometer with **void calLSM9DS0(LSM_properties* prop);**, the bias is stored in LSM_properties, in the fields float abias[3] and float gbias[3].

We write functions to read the accelerometer, the gyroscope, the magnetometer and even the temperature. They are : **void readAccel(LSM_properties***

prop);, void readMag(LSM_properties* prop);, **void readTemp(LSM_properties* prop);** and **void readGyro(LSM_properties* prop);**. These functions write the read values in the fields of LSM_properties, in ax, ay, az, gx, gy, gz, mx, my, mz and temperature. However, these are raw values. We use functions to do the conversions : **float calcGyro(int16 gyro);**, **float calcAccel(int16 accel);** and **float calcMag(int16 mag);**.

All these functions are just reading and writing in registers. The register description can be found in the documentation. Finally, we write functions to configure interrupts. These interrupts can be on pin INTG for the gyroscpope and on pins INT1XM and INT2XM for the accelerometer and the magnetometer. The functions were **void configGyroInt(uint8 int1Cfg, uint16 int1ThsX, uint16 int1ThsY, uint16 int1ThsZ, uint8 duration);** and **void configTapInt(float threshold, int8 duration);**. They take configuration (int1Cfg is what goes in the register INT1_CFG_G), thresholds and a duration for the interrupt. The tap interrupt is on INT1XM.

We connect the LSM9DS0 to the MeshBee (3.3V, GND, SDA, SCL and INT1XM) and use the interrupt handler we define (see next part).

4.5 A Driver for LDC1614

The LDC1614 is an inductance to digital converter with a I2C interface. We can for instance detect a hand which gets close to the sensor. The driver can be found in the **src/LDC1614.c** file. It is easy to use : there is an initialization function **void init_LDC1614(void)** which initialize the LDC1614 which default parameters which are used in the datasheet. Then, we have two functions, one to read sensor 0 : **uint32 read_sensor0()** and one to read sensor 1 : **uint32 read_sensor1()**.

It is important to notice that the CONFIG register must be written at the end, after all other registers are set. To modify to parameters, we must first leave the active state of the LDC1614 and go to sleep mode for instance.

Parameters can be changed directly in the initialization function but further developments may include a structure which is passed to the initialization function and which tells how to configure the LDC1614.

5 Communications

This interesting part of the project is to be able to make the sensors communicate. With Zigbee, communications are made really easy and the network can be expended.

5.1 The Mesh Architecture

With Zigbee, different kind of architectures for the network are possible. In order to allow the network to spread in all directions, we choose to work with the Mesh architecture.

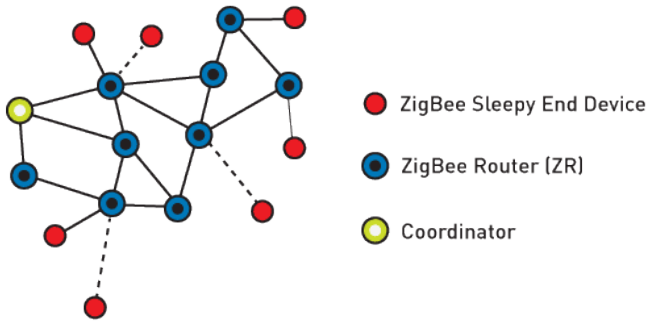

Figure 13: Mesh Architecture

There are three kinds of nodes :

- **A *unique* coordinator** : it is in charge of organizing the network.
- **Router** : it is able to propagate the network and is an access point to the network.
- **End-node** : it is only able to connect to the network.

As we want all the nodes to be the same, we choose to have one coordinator to organize the network and all other nodes are routers. This way, the network can take any configuration possible.

5.2 Easy Data Packets

It is very easy to send data packets across the network.

```
#include "firmware_at_api.h"
#include "firmware_api_pack"
//Those two include contains the function
//to easily create tsApiSpec and send them

tsApiSpec apiSpec;
uint8 tmp[sizeof(tsApiSpec)]={0};
// Will contain the final result
//For now we put in it a string
sprintf(tmp, "TEST %d", 3);
// Just a formatted string (like in printf) in tmp
PCK_vApiSpecDataFrame(&apiSpec, 0xec,
                      0x00, tmp, strlen(tmp));
// This function create a DataFrame
uint16 size = i32CopyApiSpec(&apiSpec, tmp);
//We actually create the array containing
//the API Frame in tmp
API_bSendToAirPort(UNICAST, 0x0000, tmp, size);
// We send the message to 0x0000, the coordinator
```

We begin by writing data into a variable. Then, we create a Data frame with **PCK_vApiSpecDataFrame(&apiSpec, 0xec, 0x00, tmp, strlen(tmp))** which takes as arguments a pointer to the API structure, an identifier for the frame, options for the data frame, the actual data and its size. At that stage, we have a structure which represent our frame.

To make the actual frame (with all bytes at the correct place), we need to use **i32CopyApiSpec(&apiSpec, tmp)** which takes a pointer to the structure and an array which will contains the frame. It returns the final size of the frame.

Then the packet can be sent with **API_bSendToAirPort(UNICAST, 0x0000, tmp, size)**. We can either choose to BROADCAST or UNICAST a message. Then, we can specify the receiver (0x0000 is the coordinator). Then, we give the frame and its size.

5.3 A Simple Protocol

To make the data we send easy to read, we need to specify how data are organize. At the end, a data was composed of three parts :

- What kind of value the sensor read (temperature, acceleration, humidity,...) on four bytes
- A time stamp on eight bytes (time in milliseconds, see later)
- The measurement on four bytes

As it has always the same structure, we simplify the sending of data by writing a function in **src/utils_meshbee.c** : **void send_frame(char* name, int data)**. The function takes a string of size 4 for the name of the measure and the data read. Then, it sends to the coordinator the formatted frame.

5.4 AT commands

It is also possible to communicate with AT commands and to execute a function linked to it, even on an other device. For example, to set the time, we use an AT command which is called from the coordinator to all other nodes. To do so, we need to build a specific frame.

To call a local command, we do :

```
char * aj = "ATMF500";
API_i32AtCmdProc(aj, 7);
```

This code set the loop interval to 500ms. We use the function **API_i32AtCmdProc(aj, 7);** which takes the command and its length. It would be also possible to call a remote AT command. Later, we will explain how the server calls these commands.

5.5 From Zigbee Network to GUI

Communicating inside the network is quite easy. However, we need a way to show the information that are gathered by the coordinator. To do so, we use a Raspberry Pi which is connected to the coordinator and which controls the network and display measurements.

5.5.1 Configuring the Raspberry Pi

We use a Raspberry Pi 3. It is powerful enough to also run a server (see later) and have a Wifi chip integrated. For the OS, we use Raspbian. The first thing we have to do when we is to install the OS. We use a 8Go SD card (less is not enough for the full version of Raspbian). We follow adafruit Raspberry Pi tutorial for example to know how or we search on Google. During the configuration, when we are in raspi-config, we have to remember to activate the serial communication in the advanced parameters.

We need to configure the Serial port to communicate with the MeshBee. We can find the serial port in /dev/ttyS0 on Raspberry Pi 3 or /dev/ttyAMA0 for older versions. The problem is that by default, the raspberry is configured to have a terminal on this port. So we need to remove it. To do so, we open the file /boot/cmdline.txt :

```
sudo nano /boot/cmdline.txt
```

and we remove the part with ttyAMA0 or serial0 (console=serial0, 115200). We have something like :

```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline rootwait
```

(no serial0). What follows is for the Raspberry Pi 3. We need to say that we activate UART. For that, we have to open the file **/boot/config.txt**

```
sudo nano /boot/config.txt
```

and to add the lines

```
enable_uart=1
core_freq=250
```

For the version of Raspbian we use, we need to change the core frequency. It may be corrected in future version. We have problems with the communication with the MeshBee. So, by taking an oscilloscope and measuring the baud rate, we measure 72000 instead of 115200.

With the complete version of Raspbian, we do not have to install git and python libraries to manipulate the GPIOs. We clone our Mesh_Bee repository :

```
git clone https://github.com/Aunsiels/Mesh_Bee.git
```

5.5.2 Connecting the Coordinator with the Raspberry Pi

Obviously, to communicate with the MeshBee via the serial port, we need to link it to the Raspberry Pi. Here is how the pins are organized on the Raspberry Pi 3.

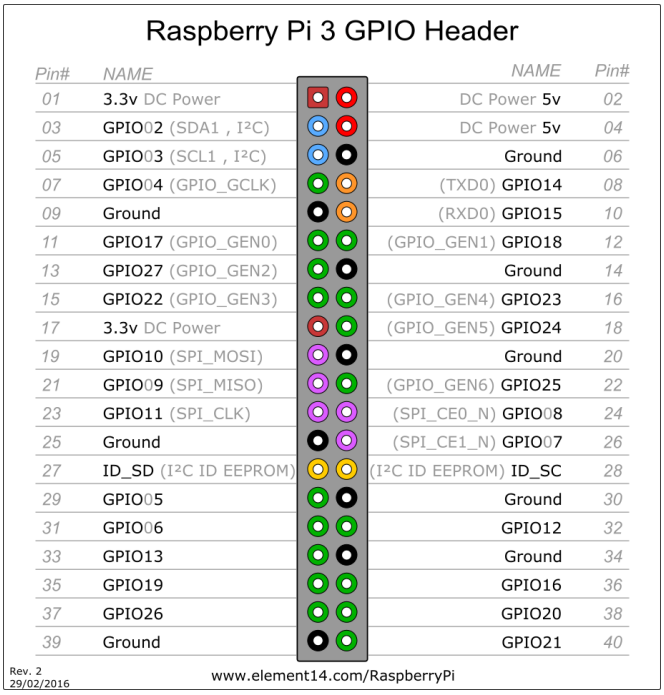

Figure 14: Pin layout Raspberry Pi

To give power to the MeshBee, we connect the 3.3V pin of the MeshBee to the 3.3V pin of the Raspberry Pi (pin 01). We do the same for Ground (GND) which is on ports 06 or 09 for example. Then, and it is important to do it correctly, to connect the TX1 pin of the MeshBee to the RXD0 pin of the Raspberry Pi (pin 10) and the RX1 pin of the MeshBee to the TXD0 pin of the Raspberry Pi (pin 08). We have something like that :

We add a LED (and a resistor !) connected to the ASSOC pin of the MeshBee to be sure that the MeshBee is connected and working.

We have the MeshBee connected, we try to communicate with it. For that, we use the software cutecom. It is easy to use. We install it :

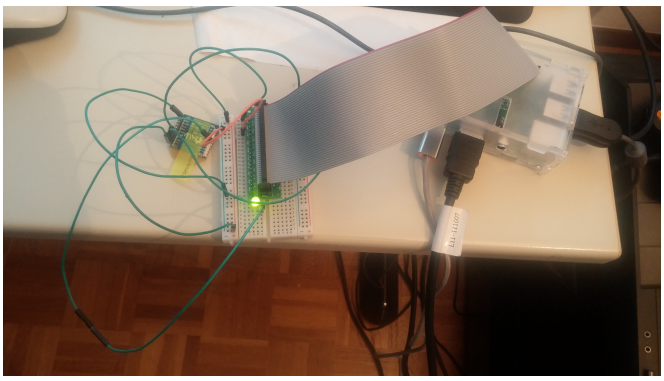

Figure 15: Photo Connections

```
sudo apt-get install cutecom
```

Then we open it by typing cutecom in the terminal.

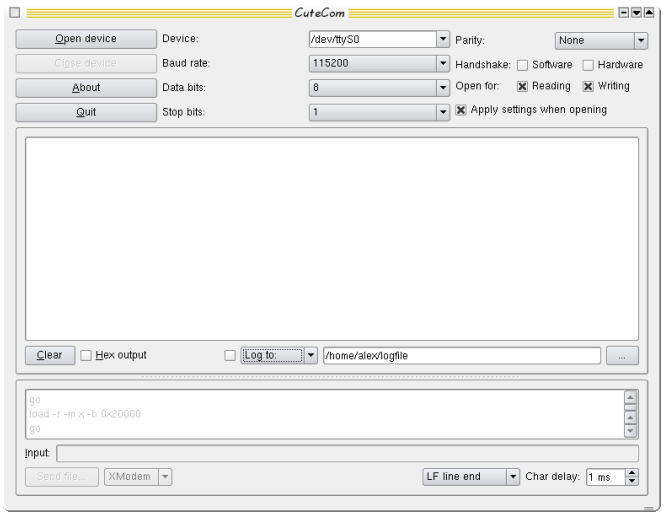

Figure 16: Cutecom

Parameters are good when we open it. We just check that the device is /dev/ttyS0 and at the bottom that we have CR line end. We then try to send AT commands for example like we saw above.

5.5.3 The Python Script

At that stage, we are sure that we are communicating with the coordinator. We need to automatize the communication and to write a program which can make

the connection between the MeshBee network and the server side. The script is **Mesh_Bee/rpi_script.py**. A MeshBee is represented by a class, MeshBee.

The first things we want to do is to be able to change the mode of the coordinator to be sure that we are in the right one. To begin with, we need to initialized the serial connection, when a MeshBee object is created. We use the serial library, which is install be default in the full version of Raspbian. The baud rate is still 115200, and we set the timeout to 1, to be sure we do not get stuck at some point of the program. Then, to facilitate the communication, we use a buffer over the serial. So, to write, we use **self.meshbee.write** with **self.meshbee.flush()** (this is very important if we want to actually send). To read, we have self.meshbee.readline or simply **self.meshbee.read(n)**, where n is the number of byte to read.

To know if we did change the mode, we check what the MeshBee answers. To be sure that there is no old data, we need to empty the buffer before. So, we set the timeout to 0 and read until there is nothing left. We write three functions, **to_at_mode()**, **to_data_mode()** and **to_api_mode()** to either go to AT, DATA or API mode.

The first experiment we do is only using AT mode and DATA mode. Actually, we use the AT to retrieve information about the network and DATA mode to read the messages. We write two functions for the information : **print_information()** which prints information about the current node (it calls ATIF, see above) and **print_nodes()** to show the nodes connected to the network. Finally, we have a function to send a message to the entire network, **write(message)**. Then, we simply read messages in a loop.

The first protocol is very simple : 4 bytes to identify the value read, the id (the mac address) on 8 bytes and then the value read on the rest. At that time, we do not have time synchronization, so no time stamp. However, the mac address is also contained in the DATA frame, we do not have access to it yet.

The problem is that changing the mode can be long (up to a second) and moreover, we lose data by changing the mode. In fact, we want to use the ATLA command to check which nodes are connected to the network (this information can be useful for the user). So, we decide to only use the API mode. Thus, we need to construct and decrypt API frames (see above for API frame structure).

We write a general function to read a frame : **api_read_frame()**. There

could be different kind of frames : data frame, topology frame,... We recognize which kind of frame it is and then call the good function to decrypt the payload : **api_data_frame(payload)**, **api_topo_frame(payload)**,... While reading the data frame, we retrieve the MAC address, so it is removed from our custom protocol.

So, the new protocol looks like that : 4 bytes to identify the value read, 4 bytes for the time stamp and the rest for the value. We have a time stamp. To share the time, we broadcast an AT command, ATST. This is done thanks to the function **send_time()**, which sends the time in second. Finally, we write a function to request the nodes on the network : **request_topo()**. We do not have to wait for the result as we did before : each module answers with a frame which is decrypted when received. The mac addresses are stored in an attribute of MeshBee, and they are all sent to the server with **send_mac_addresses()**.

Next we change the time synchronization to milliseconds (see below), the protocol becomes : four bytes to describe the value read, eight bytes for the time stamp and four bytes for the data, which is then totally raw (whereas before it was a string). So, the size is fixed (see above). The **send_time()** function then sends time in milliseconds. WARNING : the way bytes are sent is the opposite of the way it is in memory. This is because the endianess is not the same on the MeshBee (Big Endian) and on the computer.

As the server is also on the Raspberry Pi, we just have to send our requests to the localhost domain, port 9000 (see later). Then to send all the mac addresses, we open the URL **http://localhost:9000/updatesensors?s=** , followed by all MAC addresses.

Finally, we write a main function to call all our functions. This is done in an infinite loop. It just calls **api_read_frame**, and from time to time request to read the network topology and sends a time stamp.

To run the script, we simply type **python2 rpi_script.py** in the Mesh_Bee directory.

5.5.4 Server Presentation

We write a HTTP server to retrieve information from the MeshBee network and to display it to the user. The server is written using Play Framework, with Scala. We will not explain how Play Framework works nor how to program in Scala.

To run the server, we go to **Mesh_Bee/server/bin** and type **./activator run**. Then we see the port number appear : 9000. The project compiles the first time a page is required.

The configuration of the routes can be found in **conf/routes**. The HTML files are in **app/views** and the controllers are in **app/controllers**. The main controller is **HomeController.scala**. The code for the real time printing of data thanks to Web sockets is in **MyWebSocketActor.scala**.

5.5.5 Record Sensor Data

The first communication we do with the server is with data directly in the HTTP request. So, to send new measured data to the server, we need to send, to /measuredata :

```
tosend = "http://localhost:9000/measuredata?id=" + id_sensor
tosend = tosend + "&dataType=" + data_type  + "&data=" + data
tosend = tosend + "&time=" + str(time)
```

The measured data is caught by the measureData function in HomeController.scala. Then, if the data is a special type, we do some computations with the data (for instance, for the temperature, we need to transform the raw data to degrees). Then, we create a Measure object and store it. To keep the user updated, we also send a web socket message to update the chart. Finally, we send a message to confirm we read the measure.

The measured date is first stored in a global variable, **measures**. We need to take care of concurrent accesses. Later, we will change it by a database access, using MongoDB for instance. However, for development purposes, global variables are easier and easy to clean.

5.5.6 Real-time Communication

The real-time communication is done thanks to web sockets. We handle them asynchronously, thanks to actors, in **app/controllers/MyWebSocketActor.scala**. When we create a web socket, we store it into a map, **mapping**. So, when a user connects to the page, it creates a web socket. When a new data was received, it is transmitted to all connected users.

5.5.7 Checking Connected Devices

When the python wants to tell which devices are connected, it sends the list of all MeshBee as a String. Then, we check if all devices are in the String. Then, the user web page refreshes regularly to keep the printed device list updated. This is done using JavaScript, directly in the HTML page, in app/views.

5.5.8 Charts

To print the charts, we used a JavaScript library, Highcharts. The chart was then loaded when the user clicked on a given device.

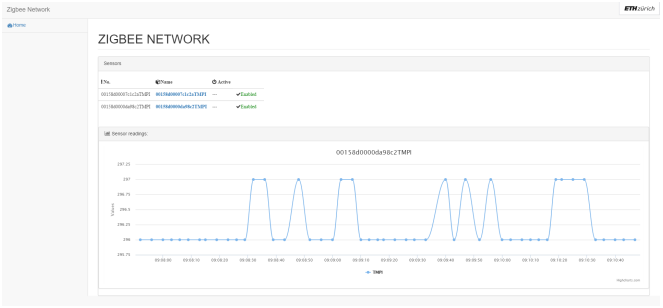

Figure 17: Charts

6 Time Synchronization

6.1 First Version

Our first try with time synchronization is to use the given ZCL library (Zigbee Cluster Library). A cluster is something defined in a norm and is a kind of container of data which can be shared in the network. So, it provides us a structure to store data and functions to interact with it. It also updates the time every second thanks to a timer and a task. However, at that time, we do not know that we need to declared everything ourselves in the configuration file. Finally, the library does nothing special as we need to code almost everything. We use the time cluster. The ZCL library can be found in **Jennic/Components/ZCL**. We have to add the source files of this library to the Makefile.

The first thing we have to do is to give the time to the coordinator. To do so, we create a new AT command we call through the API frame. Creating new

AT commands is quite easy. We open the file **src/firmware_at_api.c**. Then, there are two variables : **atCommands** which is the array containing all AT command which could be called from the AT mode, and atCommands which are commands which can be called from the API mode. We create a new command for API mode, to set the time. So, we add :

```
{"ATST", ATST, NULL, API_setTime_CallBack},
```

to atCommandsApiMode. This is a structure of type **AT_Command_ApiMode_t**, which is defined as follow (in **include/firmware_at_api.h**) :

```
typedef struct
{
    const char  *name;
    //AT command name
    uint8       atCmdIndex;
    //AT command index
    uint16      *configAddr;
    //config address
    AT_CommandApiMode_Func_t function;
    //AT commands call back function
}AT_Command_ApiMode_t;
```

The name is simply "ATST" (it needs to be four chars). **atCmdIndex** is a unique number which identifies the command and is generally a constant declared in **include/firmware_at_api.h**, in **teAtIndex**. ATST has as index 0x80. The configAddr is the address of a variable which will be set to the value given after the AT command (see above to learn more about AT commands). Finally, there is the function which is called for this command. The signature and content are :

```
int API_setTime_CallBack(tsApiSpec *reqApiSpec, tsApiSpec *respApiSpec, u
{
    uint32 time;
    memcpy(&time, reqApiSpec->payload.localAtReq.value, 4);
    vZCL_SetUTCTime(time);

    return OK;
```

```
}
```

As an argument, we have the received API frame, the frame which will be the answer and the variable which was assigned before and which contains the value of the AT command. We use the memcpy function in order to copy the time stamp in the AT command. Then we use the **vZCL_SetUTCTime** function which takes the current time in milliseconds as an argument to set the time of the cluster. To be able to use this function, the time cluster has to be initialized. We do that in the initialization of the node, in **src/zigbee_node.**, in the function **node_vInitialise**. We use the function **teZCL_Status eZCL_CreateZCL(tsZCL_Config config);**. This function receives a configuration to initialize the cluster. This configuration contains for instance the callback method for the time cluster, but also other useful information. In the callback method, **cbZCL_GeneralCallback in zigbee_node.c**, we catch some events that can happen. Among them, there can be event to request the ZCL mutex.

It was important to have mutexes when we have interrupts and concurrent accesses. Using mutexes allows us to defined Critical Sections. Critical section is a part of the program which prevents the current task to be pre-empted by an other task from the same mutex group (linked to the same mutex in the configuration file), even by a task of higher priority. To enter a critical section, we use **OS_eEnterCriticalSection(name_mutex);** and to leave it : **OS_eExitCriticalSection(name_mutex);** in vLockZCLMutex and vUnlockZCLMutex (zigbee_node.c).

Then, we have to call the Event Handler of the time cluster every second. For that, we have to configure a software timer, **APP_ZclTimer**, linked to a task, **APP_ZCLTask**, which belongs to the ZCL mutex group. In **APP_ZCLTask** (in zigbee_node.c), we call the event handler, vZCL_EventHandler, which takes a tsZCL_CallBackEvent as an argument. We put in that structure the kind of event, here E_ZCL_CBET_TIMER.

Then, we have a system to count the time in seconds.

6.2 Second Version

Counting in seconds was too restrictive for what we want to do. So we decide to rewrite all the synchronization part to be more flexible and count in milliseconds.

The first thing we have to change is to use uint64 instead of uint32. This way, we avoid overflow. In the meanwhile, we also changed the SULI library to also use uint64. The new time synchronization code was written in **src/time_sync.c**.

The use is almost the same. There is a function to initialize the variables of the time synchronization : **void init_time_sync(void);**. Then, it is possible to set the low part and the high part of the timestamp with **void setHighTime(uint32 time);** and **void setLowTime(uint32 time);**. We have to separate the lowest part and the highest part because for the AT commands, we are only allowed to transmit four bytes of date, half of a uint64. Then, we are able to check whether the time is synchronized or not with int **timeHasBeenSynchronised(void);** and finally, we can get current time with **uint64 getTime(void);**. All in all, we replaced all previous functions.

However, we erase the timer part as we do not need to count seconds anymore : we use the SULI **suli_millis** function to keep track of the time. We still use mutex as we have concurrent accesses.

As we separate the highest part and the lowest part of the timestamp, we need to do two different AT functions to set the time. ATST is replaced by ATSH (for Set High) and ATSL (for Set Low). The way it works is exactly the same, except that it calls the two functions to set the time.

Once we have done that, the communication protocol changes. A frame is now composed of four bytes to describe the value read, eight bytes for the timestamp and four bytes for the data, which is then totally raw (whereas before it was a string). So, the size is fixed.

6.3 Correcting the Clocks

Once we have the framework to synchronize the time, we want to check what is the error of time between all nodes of the network and the server, and how it evolves with time. At first, we are resynchronizing the time every minute, so the error was low, some milliseconds. However, we want to try to only synchronize time once or a limited amount of time and to see how the system evolves. The first experiment we carry out is to compare the time between two MeshBees. Both of them are connected to the Raspberry Pi via wires and Zigbee. The MeshBees synchronize their internal time with the server at the beginning of the experiment. Then, the server sends at regular intervals signals via the wire which

trigger an interrupt which sends the current local time. When we plot the time difference between two MeshBees, we observe a drift.

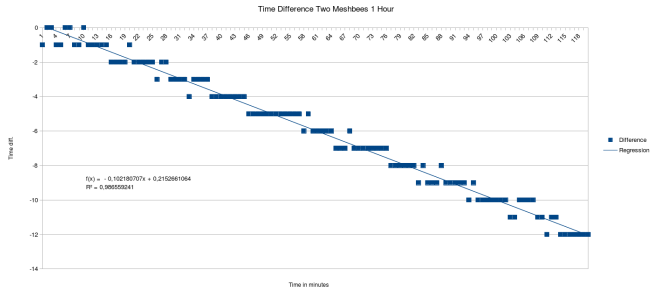

Figure 18: Two meshbees drift

(the time step is 30s) We observe a small drift, of around 0.2 ms per minute. The error can become too big with after a lot of time without synchronization but synchronizing once every 45 minutes give us an error of less than 10ms. Next, we try to see the drift of a MeshBee with the server.

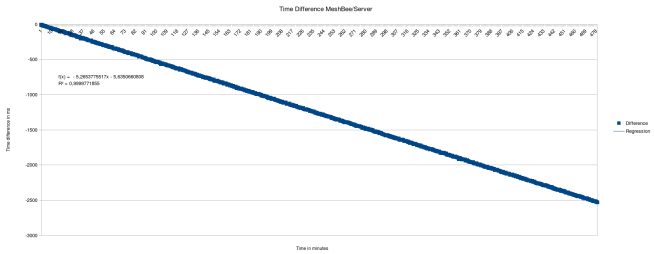

Figure 19: MeshBee Server Drift

We observe a huge drift, more than 5 ms per minutes. At first, we did not really understand where it was from. However, as it is very linear, we try a simple linear regression to correct it.

We have some noise due to Zigbee communication but the standard deviation was less than 10 ms. There is almost no drift : 0.012 ms per minutes, even if it is hard to say with the noise. The problem with this approach is that we need to record some samples before being able to correct the read values. To remove this problem, we have to understand where the drift problem is from.

We first think it is because of a stop of the timer due to interrupts, but this is very unlikely because we use a hardware timer. We check that by using an oscilloscope and measuring the clock frequency directly. There seems to be a

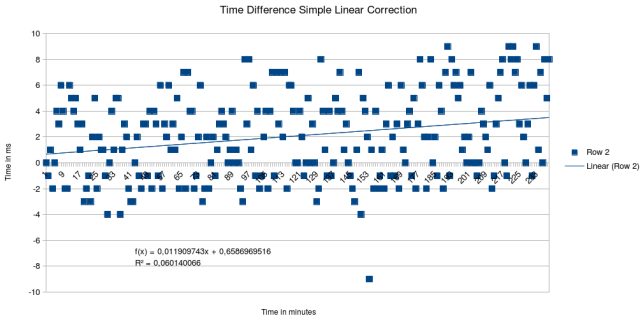

Figure 20: Linear Regression Correction

small problem here. The oscilloscope is not precise enough to be exactly sure, but it seems than the MeshBee clocks are too fast. This hypothesis was not directly considered because an external clock was used and it is supposed to be very precise. However, by reading the documentation, we learn that the clocks have a frequency tolerance of more or less 40ppm (0.004%). These constraints come from the constraints we have on UART and Zigbee communications. So the 32MHz external clock has a frequency of 32MHz more or less 128kHz, the peripheral frequency (main clock divided by 2) is 16Mhz more or less 64kHz and after the prescaler we use to count time (divided by 16), we have a frequency of 1Mhz more or less 4kHz. So, at the end, in the worst case, we have an error of 4ns every microsecond. So, after one second, we could have an error of 4ms. We measured something smaller but it could be worst.

We explore the possibility to do a calibration of the clock the first time the MeshBee is used with a wired connection. To do so, the MeshBee is connected to the Raspberry Pi with four wires : VCC, GND, a synchronization enable signal (D18) and a clock signal (D1). The clock is generated with a PWM by the Raspberry Pi. A script is used : **synchronize.py**. Then, the MeshBee catches the interrupt and compare the PWM frequency (which is supposed to be known) and its internal time counter. The correction is computed (in **utils_meshbee.c**) and saved permanently in the EEPROM memory by using the PDM library (see in **time_sync.c**).

We observe that the drift is reduced and that the noise is still here. There seems to be still a small drift but like for the linear regression correction, it is hard to estimate it on a short time interval. The advantages here are that we do not have to compute coefficients by receiving time steps. We only need the first time

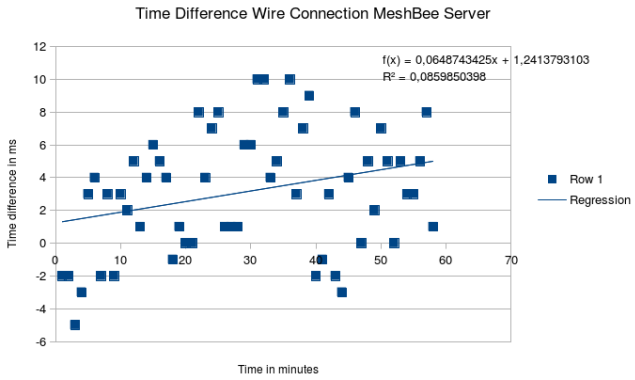

Figure 21: Wire Correction

stamp to give a good estimate of the future times. However, we cannot correct the bias there can be with only one time stamp. In fact, this bias is due to the noise in Zigbee communication, which makes the first value not centered. However, this bias could be estimated online with similar techniques we used for the linear regression, but we would lose some advantages of the wired method. Another advantage of wire synchronization compared to the linear regression approach is that it does not depend on the size of the network. In big networks the noise will be higher and so linear regression less precise. However, the bias of the wire method will also be higher.

7 Summary and Contributions

During the semester project, a flexible and scalable network of sensors was design. We proved that any kind of sensor connected by I2C can be easily integrated to the system. Moreover, using HTML pages makes it very simple to visualize data because of the high number of resources available and because nowadays, all phones have a web navigator integrated. At the end, we also partially solved the problem of time synchronization by analyzing different methods to compensate drift.

7.1 Future Work

The project can be mostly extend in the use cases. We have created a easy-to-use framework with a lot of explanations on how to use it. Choosing what to do with it is the next step. It could be use for players monitoring in a sport team, smart home, smart city,...

On a more technical side, improvement can be done. The python script should be rewritten in C. On the server side, there is work to do with web design and how to display data in a meaningful way. On the hardware side, specialized design could be done, depending on the use case. As the design of MeshBee is open source, it can be easily modified.

Epilogue

I hope you found in these pages the starting point of a brilliant idea. Do not hesitate to continue what I started and never sensor yourself. #blague