
ArdBeeScale Project

CONTENT

1	PROJECT PRESENTATION & TECHNOLOGICAL CHOICES	3
1.1	SYSTEM PRESENTATION	3
1.2	DEVELOPMENT ENVIRONMENT : PLATFORMIO	4
1.3	MICRO CONTROLLER CHOICE	5
1.4	DATA TRANSMISSION: LoRa	7
1.4.1	USING AN EXISTING GATEWAY	7
1.4.2	USING A NEW GATEWAY	8
1.4.3	GATEWAY CONNECTION WITH A 4G KEY	9
1.4.4	SETTING UP A NODE ON TTN NETWORK.	9
1.4.5	PAYLOADS DECODING	10
1.5	ENERGY	10
2	SOFTWARE	11
2.1	MICRO CONTROLLER SOFTWARE	11
2.1.1	LoRa LIBRARY CHOICE	11
2.1.2	LMIC-NODE INSTALLATION	11
2.1.3	HX711 LIBRARY	13
2.1.4	BME280 LIBRARY	14
2.1.5	LIBRARIES FOR THE DS18B20 TEMPERATURE SENSOR	15
2.1.6	PAYLOADS FORMATS	16
2.1.7	MAIN SKETCH –UPLINKS AND DOWNLINKS PROCESSING	18
2.1.8	MAIN SKETCH – POWER MANAGEMENT	18
2.1.9	MAIN SKETCH; TIME MANAGEMENT	19
2.1.10	DEALING WITH UNEXPECTED HANGS THANKS TO THE INTEGRATED WATCH DOG	20
2.2	JS CODE FOR PAYLOAD DECODING (TO BE LOADED ON TTN CONSOLE)	20
3	INTEGRATION WITH GOOGLE SHEET	21
3.1	MQTT	21
3.1.1	PROTOCOL	21
3.1.2	MOSQUITTO	21
3.1.3	MQTT & PYTHON	23
3.2	PYTHON CODE RUNNING ON RASPI SERVEUR	24
3.2.1	PROGRAMS & DATA ORGANIZATION	24
3.2.2	PYTHON CONFIGURATION FILE	25
3.2.3	RETRIEVING DATA FROM MQTT AND TEMPORARY STORAGE	25
3.2.4	DATA STORAGE IN A GOOGLE SHEET FILE	26
3.2.5	INSTALLATION	28
4	BLYNK	29
4.1	GENERAL PRINCIPLE	29

4.2	BLYNK SET-UP	29
4.3	PROGRAMMING	30
4.4	BLYNK INSTALL ON RASPBERRY	32
4.5	CONFIGURATION OF BLYNK APP	32
4.6	UPDATE AS OF FEBRUARY 2023	33
5	NODE-RED – INFLUXDB – GRAFANA INTEGRATION	33
5.1	PRINCIPLE	33
5.2	NODE-RED	35
5.2.1	FETCHING UPLINKS IN NODE-RED	36
5.2.2	DATA STORAGE IN INFLUXDB	37
5.2.3	SENDING DOWNLINKS	38
5.2.4	NODE-RED DASHBOARD ON A SMARTPHONE	39
5.3	VISUALIZATION WITH GRAFANA	39
6	HARDWARE & WIRING	40
6.1	LOAD CELL & SUPPORT FRAME	40
6.2	WIRING OF THE MICRO CONTROLLERS AND ASSOCIATED CONNECTORS	41
6.2.1	WIRING OF BME280 SENSOR	41
6.2.2	WIRING OF THE HX711 BOARD	42
6.2.3	WIRING OF THE RELAY BOARD	42
6.2.4	MICROCONTROLLER WIRING	42
6.2.5	MODIFICATION OF THE BATTERIE ENCLOSURE	48
6.2.6	CHARGING THE BATTERY	48
6.2.7	CONTENT AND CHOICE OF THE ENCLOSURE	48
6.2.8	MOUNTING THE ENCLOSURE ON THE SCALE	50
7	WEIGHT MEASURE ACCURACY	50
8	HOW TO ACCESS A GOOGLE SHEET FILE FROM PYTHON SCRIPT	51
9	WEIGHT TEMPERATURE COMPENSATION	51
9.1	PROBLEM STATEMENT	51
9.2	PROPOSED NONLINEAR MODEL	53
9.3	IMPLEMENTATION	57
10	POWER SAVING UNIT	58
10.1	INTRODUCTION	58
10.2	ABOUT THE HARDWARE	59
10.3	SOFTWARE	60
10.4	UPLINKS SENT BY THIS ENERGY SAVING DEVICE:	62
10.5	SETTING UP THE TIME OF THE ENERGY SAVER UNIT.	62
10.6	MAKING SURE EVERYTHING IS IN SYNCH...	63
11	- LIST OF FILES	64
12	- SETTING-UP THE SYSTEM...	65

1 PROJECT PRESENTATION & TECHNOLOGICAL CHOICES

1.1 System presentation

Main features of « ardbeescale » :

- Hive weight measurement with a Bosche H40 load cell
- Temperature, pressure and humidity measurement with a BME280 sensor
- Internal hive temperature measurement with a DS18B20 one wire sensor
- Data transmission on the LoRaWAN TTN network at a configurable frequency from once per minute to once per 24H
- Use of an Arduino low consumption microcontroller (Adafruit LoRa Feather M0, based on SAMD21 processor), powered by two 18650 batteries (3.7V) rechargeable on an external USB plug.
- The device includes a relay and the ad hoc hardware & software needed to remotely trigger a syrup pump from a smartphone

TTN's LoRaWAN network is widely present in my region (Grenoble/France, the home of Cycleo, the company that invented LoRa technology, which Semtech, a US company bought in 2012) and therefore in many places in this region it is possible to install a LoRa based hives using existing gateways. However my apiary happens to be a bit too far away from the closest gateway and I have therefore chosen to add one, which is connected to the Internet through a 4G key and powered by a 12V battery connected to two PV panels.

To retrieve the data emitted with LoRa protocol by the hives, I use a raspberry pi that communicates via the internet with the TTN network. Two types of integration have been developed:

- Through two python programs that run on the raspberry pi
 - `getdataFromUplink.py` which retrieves (thanks to the MQTT protocol) the data sent by the hives to the TTN server and places them in a directory of the raspberry server
 - `storedata.py` which permanently checks this directory and stores the data that have been placed in it (by `getdataFromUplink.py`) in a google sheet file in which they are visualized on charts. These programs are designed to manage a maximum of 6 connected hives per apiary on a Google sheet file.

This integration had some reliability problems that seem to be related to the availability of the google sheets files (I didn't manage to implement the error checks that could allow to totally get rid of these problems and it turns out that the `storedata.py` program crashes from time to time, about once a month for a reason that I failed to identify)

- Based on the limitation of the google sheet integration, another one has been developed using a standard IOT stack: the data are retrieved using the MQTT protocol, using an MQTT node on nodeRed that listens to the TTN broker. Another node of nodeRed stores the data in an influxDB "time series" database and the data are visualized using Grafana.

The data transmitted from the hive to the server (uplinks) are weight, external temperature, internal temperature, humidity, pressure and battery voltage, all on 8 bytes. A last byte is added to secure the transmission of downlinks: it contains the value of the last downlink received allowing to check that this downlink has been received and processed.

The data transmitted by the server to the hive as downlinks allows :

- To change the frequency of measurements
- To trigger the pump feeding the hive with sirup
- To hold measurements for a certain time in order to avoid false measurements during hive manipulation which otherwise would pollute the weight curve of the corresponding hive.
- To reset the value of the last downlink received by the server
- To set the time of the micro controller in order to be able to take measurement at precise moments. This is needed because in winter there is not enough energy to power the gateway permanently; a special device (`hso-energy saver`) has been developed to switch it ON at

precise moments and the hives should transmit data at the same time; hence the need to set the same time on all devices). Details in [2.1.9](#)

The enclosure includes following inputs / outputs:

- Three female Jack connectors :
 - Two that are very close to each other (mounted in parallel) allow to bring in and out (to the next hive) the 12V necessary for powering the pump
 - One is apart and brings 12V to the pump
- A micro USB connector (connected to the one of the Adafruit board itself) is available on the enclosure. It allows to recharge the batteries (because the board includes a charger) and to reprogram the processor without having to open the case.
- A push button triggering the pump in order to easily calibrate it.
- There is no connector for either the BME280 sensor or for the load cell: corresponding cables pass through the housing.

The enclosure also includes an on/off button which is needed to easily reprogram the device without opening it (reprogramming the device is possible only during program start-up; later on the device is almost all the time in sleep mode and would not respond to the serial port)

The firmware also includes a temperature compensation model that reduces fluctuations in weight measurements when temperature changes. See [chapter Error! Reference source not found.](#)

1.2 Development environment : platformIO

platformIO seems to be a quite powerful and widely used development tool, the most common alternative being Arduino IDE which is deemed to be less feature full and does not have a debugger. The installation procedure of platformIO can be found [here](#). There is also a good and clear explanation video [here](#) including explanations in text form.

When programs developed with Arduino IDE are used (.ino extension), it's needed to convert them into platformIO file (.cpp extension). See explanations in the platformIO doc [here](#). It is the same code, the main difference being the order in which functions are declared.

It seems to be possible to use Python on platformIO with Arduino (by charging the ad hoc extension) but I haven't tried. Documentation is [here](#).

To use platformIO, Visual Studio Code has to be loaded first and then the PlatformIO plug-in (by clicking on the "extension" icon, type platformIO and it will load automatically).

A good video tutorial from a Canadian guy is available [here](#).

In platformIO, different projects are stored in different sub directories of `/platformIO/Projects` on the Mac. You can open several projects at the same time in the same workspace. To add a project in a workspace select "add folder to workspace" (if you just do "open folder" it opens the project but it closes everything else, including the current workspace). To remove a project from the open workspace select "remove folder from workspace".

You can save a workspace containing several projects. Re opening it opens all the projects.

Sometimes and for some reasons, I met difficulties in loading my projects in PlatformIO (impossible to add a folder). I resolved this by clicking on the small "home" icon and then could choose the project to be loaded... I need to dig more in PlatformIO to fully understand how it works...

Libraries installation :

Click on the home icon, then choose "library". The library is installed only for the project that needs it. The library is added in the `.pio` directory (therefore not visible in the finder).

Alternatively, it's possible to add a library by adding the name of its `libdeps` (library dependency) in `platform.ini` which is project specific:

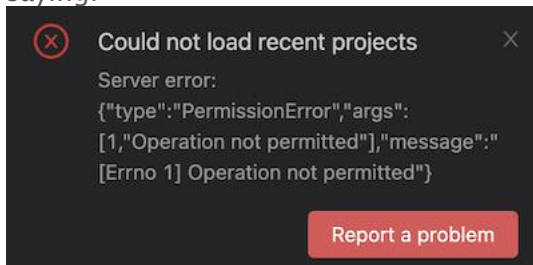
```
lib_deps =  
  fabbyte/Tiny BME280 @ ^1.0.2
```

In fact it seems that adding some libraries will automatically add this line to `platformio.ini` but in other instances, one need to load the library as described above and also add the reference to it in the file `platformio.ini` (as explained in the "installation" tab available once you have added the library using the "home" icon of PlatformIO. Need to double check as I am not fully sure...

At compilation time, the library will be loaded and it will show up in folder `.pio/libdeps`. In order to remove it, simply remove its reference in the `libdeps` and remove the directory of this library which resides in `.pio/libdeps`

Official documentation of platformio.org is [here](#).

For information I have met an issue with PlatformIO which I was unable to resolve by myself. For some reasons, all of a sudden it became impossible to list the installed libraries. Clicking on the home icon and then on Libraries et selecting "installed libraries" triggered a pop up error message saying:



I posted a question about this issue on PlatformIO community forum and a PlatformIO support guy instructed me to type the following command in the system terminal:

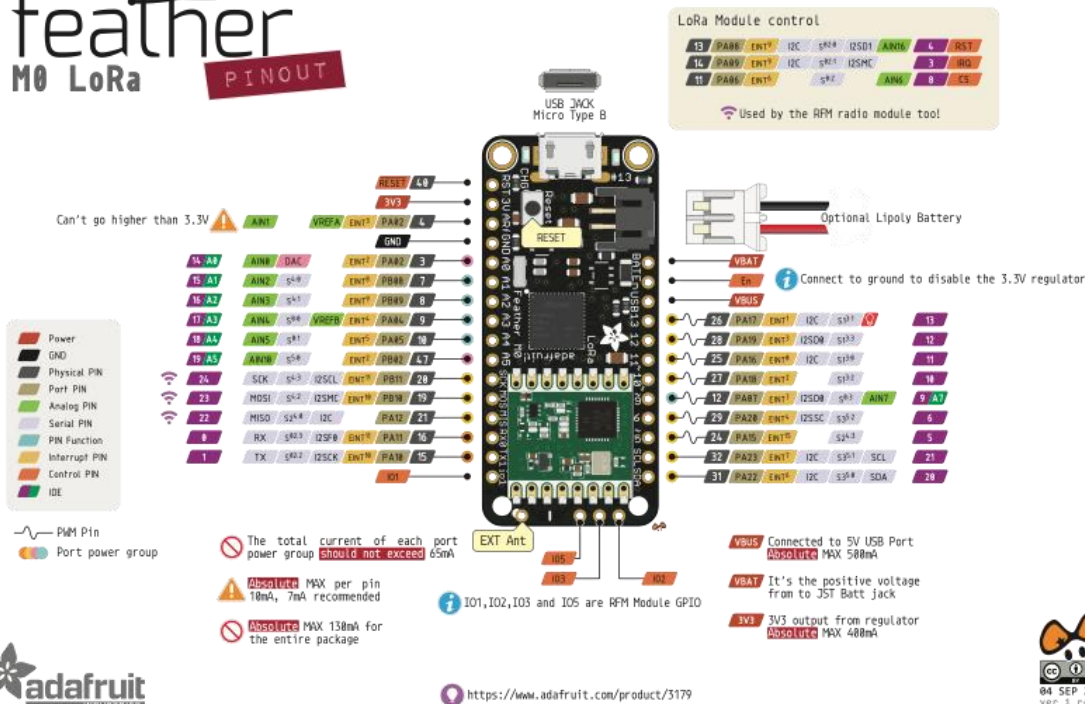
```
rm ~/.platformio/homestate.json
```

It resolved the problem but I completely fail to understand why and what happened.

1.3 Micro controller choice

Initially I started with a [Moteino](#) which is a very small size Arduino compatible board (ATmega328p processor), proposed par [lowpowerlab](#). This Moteino board has a piggy back LoRa option, connected to the SPI bus. The supplier is in the US (knowing the radio module comes from China) but there is now a distributor in Germany (Welectron) which is selling Moteino in Europe with the a European frequency option (RF95).

But in the end, because Moteino turns out to be a 8bits processor with not enough memory for the "standard" LoRaWan library (only the old version of the library which is not supported anymore would work) I had to drop this board and I finally selected [Adafruit Feather M0 with RFM95 Lora Radio - 900MHz](#) (which can work at 868MHz, the authorized frequency in Europe). I paid 51€ttc on Amazon. Documentation is [here](#). This board, which is Arduino zero compatible, is powered with 3.3V and is based on an Atmel 32 bits ARM ATSAMD21G1 processor. It carries much more memory than Moteino and includes a micro USB interface with the ad hoc charger circuitry allowing to recharge the battery by plugging a micro USB cable connected to a power source. It also includes a battery connector.

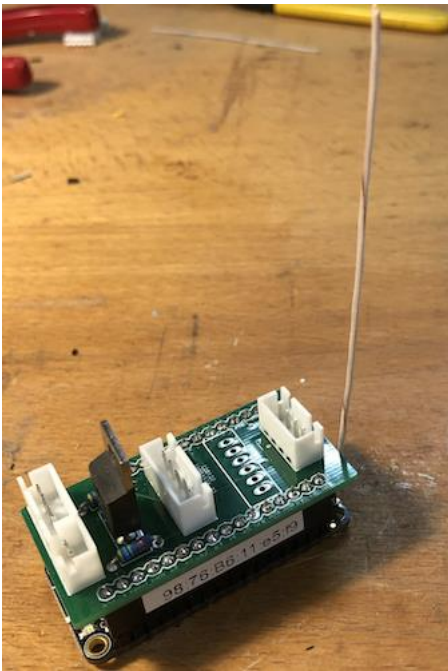


This board can be powered with a 5V power supply connected to micro USB plug; the internal circuitry will convert the voltage to 3.3V.

There is an HX711 (A/D converter for the load cell) driver available for this board (<https://github.com/bogde/HX711>) which is the one used by hiveeyes.

Adafruit busIO driver must also be loaded in order to be able to use the I2C bus, which needed for the BME280 device (temperature, humidity and pressure sensor).

For the antenna, I initially bought one on Amazon for 11€ but when testing this antenna I found that I lost 10 to 20db on the RSSI of the received signal compared to an antenna made of a simple wire of $\frac{1}{4}$ of the wavelength. So I abandoned the idea of an SMA antenna and just used a simple wire that I tilted a bit to fit into the box. To make something more sophisticated, the best thing to do is to refer to [this forum](#) which explains how to make a half wavelength dipole antenna using the [right cable](#) of the right length to connect it to the board.



This is a picture of the module (with add-on board plugged in, as explained later in this document) equipped with a $\frac{1}{4}$ wavelength antenna (8.62cm):

1.4 Data transmission: LoRa

It was a Grenoble-based company that developed LoRa (Cycleo, created in 2010), bought by Semtech two years later for \$5M... The LoRa alliance was created in 2012.

One way to become familiar with the technology is to read the [this page](#) presenting what they call "the things node" which is a simple device allowing to make tests. This same page has a menu that presents other devices.

LoRa protocol is close to the hardware but there is an associated network protocol, called LoRaWAN.

The web site www.thethingsnetwork.org manages a free LoRaWAN network. There is a « [quick start](#) » page containing multiple explanation videos and listing several popular development boards. In particular, there is a link on some videos from TTN explaining how to do the set-up of a LoRa device or a LoRa gateway and which speaks about payload formatters (for getting the data correctly formatted).

The TTN community offers a tool called "the things stack" which allows to connect to a LoRaWAN network and to manage all LoRa devices, including end nodes and gateways. The same tool allows to format the received data so that it is readable (payload formatter). On TTN's LoRaWAN network, you can connect to the network even if you don't have a gateway, any gateway on the network being usable, as long as it is accessible (=near enough).

There is very wide variety of LoRa compatible sensors of all kinds: see a list on the TTN site [here](#).

With TTN, there are limitations in the number of messages transmitted and their length (max: 30s air time/day. See TTN « [fair policy](#) »).

TTN web site has [a documentation page](#) on LoRa and LoRaWAN.

1.4.1 Using an existing gateway

TTN publishes a web page with the latest network coverage indications: <https://ttnmapper.org>

This mapping can be updated by users using a TTN tool. There is a page on the TTN site that gives all the explanations

Most cities and their suburbs are covered by the TTN network.

1.4.2 Using a new gateway

To use the hives in a somewhat deserted place (= not covered by a LoRa gateway of the TTN network), it is necessary to set up a new LoRaWan gateway and hook it up to the TTN network. See possible solutions on [this page](#). The best is probably to buy the gateway offered by thethingsindustries [here](#). There is [a thread](#) on the forum that explains how to transform the indoor gateway into an outdoor gateway. I also asked the question of choice of gateway on the TTN forum and I received [an answer](#) that seems interesting if you want an outdoor gateway.

There is also a Raspi HAT which looks interesting and is unexpensive [here](#) but I was told on the forum that it doesn't do the job correctly.

There is a [page](#) on TTN explaining how to make its own gateway and which lists all existing LoRaWan gateways.

1.4.2.1 Installing the TTIG gateway

I started by using an indoor gateway (the things indoor gateway also called TTIG). For memory, on this gateway, EUI is 58A0CBFFFE802B8B, WIFI code is wJdMxcHv and in the console I have chosen the id ttig-hs01. Installation instructions are [here](#).

Note: once the gateway has been connected to the TTN network, it is easy to change the WIFI network to which it is connected: press the set-up button for 10s (until the LED flashes orange), this will turn the gateway into a WIFI access point which SSID is MINIHUB-XXXXXX. We then need to connect to this access point (WIFI password is wJdMxcHv in my case), and go to the page [192.168.4.1](#) and click "save & reboot" after selecting the network on which we want the gateway to connect.

Attention : there is a known bug in the TTIG indoor gateway: if the gateway is requested less than 1 time every X minutes (I think X is around 15), the gateway disconnects the LoRa nodes and then refuses all the downlinks coming from them.

1.4.2.2 Installation of the Dragino gateway DLOS8 Outdoor

I also installed a Dragino outdoor gateway.

When powered on, it exposes a WIFI network called **dragino-xxxxxx** which password is **dragino+dragino**. Once connected, the gateway can be accessed at address [10.130.1.1](#)

User Name: root

Password: dragino

We can then configure the WIFI access (need to check the box "Enable WIFI WAN client" which by default is not)

WiFi

Radio Settings

Channel (1-11)

1

Tx Power (0-18) dBm

17

WiFi Access Point Settings

Enable WiFi Access Point ☒

WiFi Name SSID

dragino-20f25c

Passphrase (8-32 char)

.....

Show

Encryption

WPA2

WiFi WAN Client Settings

Enable WiFi WAN Client ☒

Host WiFi SSID

BoxSFR_Sors_ext

Passphrase

.....

Show

WiFi Survey

Choose WiFi SSID...

Encryption

WPA2

WiFi status: OK. Click Refresh to check status.

Save&Apply

Cancel

Refresh

Meaning of the gateway LEDs :

- ✓ **SOLID GREEN**: DLOS8 is alive with LoRaWAN server connection.
- ✓ **BLINKING GREEN**:
 - a) Device has internet connection but no LoRaWAN Connection.
 - or
 - b) Device is in booting stage, in this stage, it will stay **BLINKING GREEN** for several seconds and then **RED** and **YELLOW** will blink together.
- ✓ **SOLID RED**: Device doesn't have Internet connection.

Then we need to create the gateway on TTN network, using the following link :

<https://eu1.cloud.thethings.network/console/gateways>

The "gateway Id" can be found in the LoRaWAN menu of the Dragino gateway configuration page. This is also where you choose the "server address" (eu.cloud.thethings.network) and the service provider (The Things Network v3).

.

1.4.3 Gateway connection with a 4G key

Initially I tried to use a 4G key as a WIFI "access point", the dragino being supposed to connect to Internet through this access point. For some reason (which I fail to understand), the dragino refuses to connect to this access point even though it connects without any problem my home WIFI access point.

Even if I bought the Dragino DLOS8 gateway without the 4G option (the "WIFI only" version was less expensive), I realized that the product was actually including a slot for a SIM card and when I tried to plug a SIM card in it, the device quickly connected to the Internet after I correctly configured it. What a piece of luck !

1.4.4 Setting up a node on TTN network.

There are two ways to activate a node on TTN: OTAA (Over the Air Activation) and ABP (Activation By Personalization). I understand that OTAA is preferable because the corresponding protocol

includes a handshake that seems to guarantee better security. Each type of activation involves a different code on the micro controller; I have used the OTAA code.

Procedure :

1. Go to the TTN console in the "Applications" section
2. Create a new application and give it a "unique application ID".
3. Then create the new device (node). There are two options to create a new device; choose "Manually" and "Node".
4. Specify which version of TTN is supported by the library. You have to search in the library files to find this information which is in the library.properties file (1.0.3 in my case)
5. Fill frequency plan (Europe)
6. Click on "generate"; the system creates a unique deviceEUI. (EUI means "Extended Unique Identifier").
7. The appEUI (or joinEUI) is to be filled with 0's if the device you want to register does not have a specified devEUI (which is the case for DIY devices).
8. The appkey is generated by the system when you press the "generate" logo.)
9. The "end Device Id" has to be filled in with a string of your choice. Be careful when you create a device Id, deleting it does not really remove it from the TTN database and it is impossible to create a new device Id that is the same as the one you have deleted.
10. Click on « generate » so that the system creates the device on TTN

See section [2.1.2 LMIC-node](#) (text in blue) for explanation on how to update the library parameters for the newly created node.

1.4.5 Payloads decoding

The TTN console allows you to load a small javascript program that will retrieve the data received by the node and decode them by loading them into different variables that can be transmitted and retrieved by a python program running the paho library. See details in section [2.2](#)

1.5 Energy

The power of LoRa emission is apparently adjustable by software (I do not know how...)

It is indicated in the manual of the Feather M0 LoRa board that with a transmission power of 20dBm, a current of 130mA is needed during 70ms in order to transmit a payload of 20 bytes. Therefore sending such a payload requires an energy amount equal to :

$$E = 0.13 \times 7 \times 10^{-2} A.s = 0.13 \times 7 \times \frac{10^{-2}}{3600} = 2.5 \mu A.h$$

It is therefore not the sending of payloads that consumes a lot of energy even if we send them regularly. If we send 6 payloads per hour, each with 20 bytes, the energy consumed each day is 360 $\mu A.h$, meaning that a single battery could last several years.

However, the quiescent current of the micro-controller together with its radio is much higher: 13mA total, including ~2mA is for the radio itself. This is about 300mAh, meaning that a 3000mAh battery would last 10 days... It is therefore essential to put the controller in standby mode between measurements if we want to achieve a reasonable battery lifetime.

It seems that the power consumption is very variable depending on the software options chosen. See [this blog](#) which gives examples (weather station using a moteino). The main idea is to put the processor in sleep mode and to wake it up at regular interval using timer interruptions

Information on power consumption [here](#).

See also a good explanatory video [here](#) with the associated blog (with the code) [here](#). The principle consists in using the library `RTCzero.h` which uses the real time clock in order to wake-up the chip after a predefined period of sleep.

The author explains that it's possible to change the source of the RTC clock, which by default is the external clock (you can choose an internal clock) and that this saves 10 μA at the cost of a certain

loss of precision (10µA is 24mA.h in 100 days, so it is not worth optimizing by losing precision). You can find the documentation of `RTCZero` [on arduino.cc web site](https://on.arduino.cc/web/site).

The problem in our case is that we can't stop the processor anywhere without disturbing the timing of LMIC.

The only way I found to interrupt the processor without messing up the LMIC timing is to introduce the sleep at the end of the routine that handles the event « `EV_TXCOMPLETE` ». It's also necessary to delete the last line of the « `doWorkCallback` » callback function which reschedules the callback function [`os_setTimedCallback(...)`] and replace it, at the end of the code treating the event `EV_TXCOMPLETE` with a line launching immediately the callback function [`os_setCallback()`] as soon as the processor wakes-up. The sleep time of the processor is equal to the interval between two consecutive measurements as specified in the constant `DO_WORK_INTERVAL_SECONDS` in `platformIO.ini` file (in reality it will be a little bit longer because you have to add to this time the execution time of the callback function and the processing of the following events (about ten seconds in total) plus the time needed to process the downlinks if there are any (this can be quite long in the case of a downlink triggering the pump because the routine `processDownlink()` waits until the pump turns off before returning).

Power consumption calculations: the board consumes about 13mA when it is awoken and it stays awoken for about 8 seconds to send its uplinks each time it is woken up. If we send 4 uplinks per hour, the daily consumption will be :

$$E = \frac{13 \times 10^{-3} \times 8}{3600} \times 4 \times 24 = 2.77 \text{ mA.h},$$
 which is about 1A.h per annum. In reality this is higher, which means that the whole process can be improved.

2 SOFTWARE

2.1 Micro Controller Software

2.1.1 LoRa library choice

[This page on TTN](#), explains that the original library from IBM has been ported and has forked. Now the last version to be used preferably is the [MCCI LMIC](#) version, the original version [Classic LMIC](#) being a possible solution when RAM space is limited (which was the case with Moteino that I initially tested, before dropping it because the corresponding library is not supported anymore; hence the move to Adafruit board)

This library, which is to be loaded on PlatformIO or on the Arduino IDE, gives some examples of applications but it specifies that these examples are limited and suggests to use rather the work of Lopes Parente : [LMIC-Node](#) available on github. This is a complete application example, sending uplinks et receiving downlinks, using PlatformIO development environment and preconfigured for a variety of different LoRa board, including the Adafruit board that I chose. It's also a good idea to have a look at the Adafruit site which includes a section showing how to set-up their board for TTN. It's [here](#) and it includes explanation on the code and how to decode the « payloads »

Here are other possibly useful references :

- 1) I have found a Dutch person which has used the LMIC library with the same HF module with an Arduino. See [here](#). But the proposed activation method is ABP and not OTAA.
- 2) There is also a video from [Andreas Spiess](#) which seems to use the same library with an Arduino shield.

2.1.2 LMIC-node installation

Extract from [LMIC-node](#) on github :

- *Basic steps to get a node up and running with LMIC-node:*
 - *Select a supported board in `platformio.ini`.*

- *Select your LoRaWAN region in `platformio.ini`.*
- *Provide the LoRaWAN keys for your node (end device) in `lorawan-keys.h`.*
- *Compile and upload the firmware and you're ready to go!*

LMIC-node includes all the mechanics needed to send uplinks and receive downlinks so that the only code to add is the one necessary to read the sensors and create the payloads (which will be sent by the main LMIC-node sketch) and to decode the received downlinks in order to take the appropriate actions according to their contents. LMIC-node package doesn't include sleep mode so it had to be added.

LMIC-node installation is quite simple. **The procedure and the attention points are the followings :**

(the lines shown in blue are the ones that need to be modified when making a copy/paste of the PlatformIO workspace of an existing connected scale to create a new workspace for a new scale)

- A wire connection need to be added on the Adafruit board (it's explained both in [Adafruit doc](#) and in the `platformio.ini` file of LMIC-node). The connection is between DIO1 pin of the radio module and pin GPIO6 of Adafruit board.
- A project ("ardbeescale") must be created in platformIO, specifying "Adafruit Feather m0" as the board
- The file `platformio.ini` must be replaced by the one of LMIC-node
- The content of the file `LMIC-node.cpp` must be copied into `main.cpp`, replacing the existing content in this file
- The file `LMIC-node.h` must be copied in the directory `include`
- The controller board (`adafruit_feather_m0_lora` in my case) is chosen by removing the comment of the corresponding line in file `platformIO.ini` (and by commenting the line above « `<platformio.ini board selector guard>` »)
- The « board support file » corresponding to the controller board that I use must be copied in `platformio` (file `bsf_adafruit_feather_m0_lora.h`). Need to create a directory in `src` (`src/boards`) and put this file here.
- The region setting is done in file `platformIO.ini`, line 159. Default is Europe.
- In the file `platformio.ini`, the right library must be selected (`mcci-catena/MCCI LoRaWAN LMIC library`, the last version). All the `lib_deps` (libraries dependencies) that are useless in our case (`olikraus/U8g2` for example, which is a driver for OLED) and it's of course needed to add all those libraries that the system will use (`HX711` for example)
- The LoRa « keys » must be specified in the file `lorawan-keys.h` (see `lorawan-keys-example.h` on [github](#)). The content of this file must be copied/pasted in a different file name (`lib/keyfiles`). The keys include `AppEUI` (sometimes called `joinEUI`), `deviceEUI` and `appKey` ; they are provided by TTN on the TTN console when the node is created. Warning, the keys are provided on TTN console under the wrong format. In order to get the right format :
 - For `AppEUI` & `devEUI` :
 - click the <> button (1st left) to toggle between hex and C-style
 - click the second button from the left to toggle to lsb format
 - click the copy button on the far right
 - For `appKey` : same but msb first format must be kept (do not click on the button changing msb to lsb)
- `DO_WORK_INTERVAL_SECONDS` value in file `platformIO.ini` must be set in order to determine the frequency at which `processwork` loop is called.

- To send uplinks, [LMIC-node](#) uses port 10, which is not the default one. For downlinks, this is port 100.
- [LMIC-node](#) code includes a portion where the node sends an uplink containing the value of a counter, the counter being incremented at each loop. The counter can be reset by sending 0xC0 to the node with a downlink. Obviously this part of the code must be removed but it's useful to play with this example code in order to check that everything works fine.
- [LMIC-node](#) includes a constant called « `DEVICEID` » used to identify a node (especially in the status information sent on the serial port or on the display when there is one). By default this `DEVICEID` is the devid of the board as defined in the BSF (Board Support File), which is in directory `/src/board` of platformIO. In my case, the default deviceId was `feather-m0`. This default value can be overridden by defining the constant `DEVICEID` in file `lorawan-keys.h`
- LMIC-node package on github includes a javascript file [lmic-node-uplink-formatters.js](#) to be loaded on TTN in order to decode the data emitted by the node (the uplinks). This is done from the TTN console after selecting the application corresponding to this node, by clicking on the left « payload formatters » / uplinks and then on the right the option "java script" must be selected and then the code that was in the file `lmic-node-uplink-formatters.js` has to be copy/pasted with the ad hoc changes.

Warning : LMIC-node example transmits only integer data (the counter value).

Warning : LMIC-node example only transmits integer data (the counter value). Therefore the coding of these data to transmit them in the payload is very simple and on the other hand the javascript code to load on the TTN console is also very basic. If the data are of "real" type, you have to use the library's "built-in" functions that convert the real into two integers and the javascript code (on the ttn console) does the opposite. You can find examples of coding in the [Adafruit example](#) which transmits the temperature and the humidity and the javascript code decoding the payload data in the readme of the library. See the explanation in red here below

I have also tried to use the [instructions from Adafruit](#) in a different implementation (just to try):

- Need first to install the library `MCCI LoRaWAN LMIC library`
- Need to modify file `lmic_project_config.h` (in the directory containing the library) in order to select the region
- The compilation of the code `ttn-otaa-feather-us915-dht22.ino`, copied in `main.cpp` generated an error "do_send was not declared in this scope". I have added a « forward declaration » of this function on top of the code line 33 : `void do_send(osjob_t* j);`
- In the `setup()` routine, there is a line selected the « subBand ». This notion doesn't exist in Europe and therefore the line `LMIC_selectSubBand(1)` must be commented

To be noted : in the Adafruit example (explanation [here](#)), the readout of the DHT22 temperature is converted in two bytes (of type `uint16_t`, meaning unsigned integer 16 bits) with a function of the library called `LMIC_f2sflt16(x)` for « float to signed float 16 bits », `x` being comprised between -1 and +1 (in order to respect this, temperature is divided by 100). In summary, `LMIC_f2sflt16(x)` function takes a real number `x` greater than -1 and lower than 1 and transforms it into an integer of two bytes. To get the payload, it's therefore necessary to convert back those two bytes into the original real number. The javascript code supplied by Adafruit does the job (it uses the code supplied [here](#) in the readme [LMIC library](#)).

2.1.3 HX711 Library

Just for information, I have learnt on [hiveeyes forum](#) the existence of a different ADC chip which supposedly works better than the HX711: the chip NAU7802. I had already bought all my HX711 and I have therefore not used this new one.

Need to read <https://hiveeyes.org/docs/arduino/firmware/scale-adjust/README.html> There are sketches used to adjust the scale parameters and some interesting links useful to better understand load cells.

Searching « arduino hx711 library » on Google brings up [this page](#) of arduino.cc site and it seems this is the library that is used by the Hiveeyes team. The examples are self-explanatory.

Clicking « read the documentation » on the same page brings up a github page which readme points on the hiveeyes forum. There is in particular [an interesting article](#) on the measure theory.

Summary : Oddly enough, the sensitivity of a load cell is measured in mV/V (typically 2mV/V). This is the measurement of the voltage across the Wheatstone bridge per volt of excitation voltage of this bridge, when the full load of the load cell is applied. 2mV/V means that if we apply 5V as excitation voltage, we will measure 10mV on the Wheatstone bridge when the full load is applied. To measure this voltage, the ADC will first apply a gain (PGA for Programmable Gain Amplifier). If the ADC can measure between 0 and 5V and the gain is 128, this means that the range applicable to the input of the ADC is about 40mV (5/128). However, we are only going to apply 10mV, so we will only use 22 of the 24 available bits. In addition the ADC is associated with a noise which is specified at 17nV. But one bit of the ADC corresponds to 2.384nV (10mV/22bit). So we lose about 3 bits. And he continues to state all the reasons why we lose precision...

There are many HX711 libraries available in platformIO. To find the one used by Hiveeyes, you have to type "Bogde" in the search field. But I saw on [a post](#) on the Hiveeyes forum that Clemens Gruber (one of the Hiveeyes gurus) mentioned another library.

It seems that the two libraries are the same but I noted that in the [original one](#) the updates are more recent so it is the one I choose.

2.1.3.1 Usage of HX711 library

The library is not very well documented. It provides the `tare()` function which allows to tare the scale before the measurement but it does not work for a scale on which the weight is permanently applied. For that you have to use the function `set_offset(xx)` where xx is the value read on the ADC when no weight is put on the scale.

Other useful functions are :

- `read(n)` direct reading of the ADC output (average over n measurements)
- `get_value(n)` reading of the ADC output minus the tare value
- `set_scale(ratio)` to indicate to the scale the number of ADC units per gram
- `get_unit(n)` direct reading of the applied weight (= `get_value(n)/ratio`)
- `power_up()` and `power_down()` wake-up and standby of the ADC

2.1.4 BME280 library

Warning: the microcontroller documentation indicates that there is no pull-up on the SDA and SCL I/O and that it is "possible" that it is necessary to add a pull-up resistor from 2.2K to 10K. In fact, after a few hours of normal operation, the BME280 reading stopped working correctly and I added two pull-up resistors. To limit the consumption I used 50K resistors and it worked without problem. Nevertheless the corresponding power consumption is not negligible. If the bus is LOW all the time, the resistor will generate a consumption of 70 μ A, which corresponds to 0.6Ah in one year. It could therefore be useful to connect the pull-up resistors not to the 3.3V but to the output of a GPIO and to put the 5V only when it is needed.

In order to use the I2C bus, the driver has to be loaded: `#include <Wire.h>`

To verify that the microcontroller detects the sensor on the I2C bus, I used an I2C scanner found [here](#). It detects the BME280 at address 0x76. Problem : the Adafruit library waits for the sensor at the I2C address 0x77. We can either change the address of the sensor (complicated, need to tinker on the sensor) or, much simpler, change the line that initializes the code by adding the address :

Line `status = bme.begin();` must be replaced by `status = bme.begin(0x76);`

I use the Adafruit BME280 library which allows to communicate either in SPI or in I2C (simpler). You have to load this library and the one called "Adafruit Unified Sensor" (I think it loads automatically when you load the other one).

The library is written for the Arduino IDE which accepts that functions are defined after the code that uses them, which is not true with PlatformIO; it is therefore necessary to move the code of these functions to the beginning of the program.

To load the library, following lines has to be added in the file `platformIO.ini` after « `lib_deps` = » :

```
adafruit/Adafruit Unified Sensor@^1.1.4
adafruit/Adafruit BME280 Library @ ^2.2.2
```

For the program to compile it is also necessary to add in the file `main.cpp` the I2C driver (`#include <Wire.h>`) as well as the Adafruit libraries:

```
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

Then the BME280 object is created :

```
Adafruit_BME280 bme;
```

(to be added in the first portion of the code marked « user code begins here » in `main.cpp`)

Then the sensor has to be initialized in the last section of the user portion of the code, at the end of the same file, by adding the line : `status = bme.begin(0x76);`

Along with the code taking the measure.

The power consumption of this sensor is <1mA in read mode and <5uA in idle mode.

Update April 2023: when the BME280 breaks down for some reason, the Adafruit library code gets stuck waiting for it indefinitely. This not only prevents any further weight measurement to be taken and transmitted but also prevents the software to put the processor in sleep mode, which of course sink to much current and empties the battery very quickly. I have therefore changed the library and am now using the Sparkfun one.

2.1.5 Libraries for the DS18B20 temperature sensor

As explained in chapter [6.2.4 Microcontroller wiring](#), I have added a « piggy back » connector to the microcontroller board in order to be able to connect a DS18B20 temperature sensor to be used for measuring the brood temperature inside of the hive.

The DS18B20 sensor is a "one wire" sensor (using a single "data" wire in addition to the power supply), with the possibility of putting several sensors in parallel sharing the same "data" wire.

To do this, two libraries need to be added : "OneWire" and "DallasTemperature". The code is then as follows:

```
#include <OneWire.h>
#include <DallasTemperature.h>
// Data wire is plugged into pin GPIO 5
#define ONE_WIRE_BUS 5 // This line is part of parameters.h header file
// Setup a oneWire instance to communicate with any OneWire devices
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

void setup(void)
{
  sensors.begin();
}
```



```

}
void loop(void)
{
  // call sensors.requestTemperatures() to issue a global temperature
  // request to all devices on the bus
  sensors.requestTemperatures();// Send the command to get temperature reading
  Serial.println("DONE");
  Serial.print("Temperature is: ");
  Serial.println(sensors.getTempCByIndex(0)); // 0 means 1st device on the bus
  delay(1000);
}

```

2.1.6 Payloads formats

The sequence is very simple:

The program runs once every 15 minutes (this number can be changed remotely by sending an ad hoc downlink, see below) and puts the system in stand-by mode by cutting off the sensors' power supply as soon as it has finished doing what it has to do, i.e. :

1. Take a weight measure by reading the HX711
2. Take a temperature, humidity, pressure measure
3. Take a measure of the battery voltage
4. Take a temperature measure of the interior temperature of the hive
5. Pack all those measure in a « payload » to be transmitted to ttn network
6. Process the downlinks that may have been received

The program can indeed receive "downlinks" at the end of an uplink transmission.

These downlinks are sent either manually on the console or with a python program (pushDownlink.py) running on the raspberry pi used for integration. It can also be sent by using Blynk, which is a smartphone IOT application communicating with a python program (blynkMain.py) running on a raspberry pi.

These downlinks have different uses :

- Controlling the sirup pump feeding the hive with sugar sirup
- Changing the uplinks emission frequency
- Resetting the last downlink received by the node. Indeed, as LoRa communication is not 100% reliable (radio) it is necessary to check that the transmitted orders have been processed (for example the pump has actually supplied the hive with syrup). I have therefore included in the payload of each uplink the value of the last downlink received. When the verification is done, this value is reset to 0. This is the purpose of this command
- Sending a "hold measure" command, which is a time during which no measure will be taken in order to avoid polluting the monitoring graphs during the visits where the weight of the hive changes because the hive is open. We don't want to take measure at that time.

Warning: to modify the length of the uplinks sent (=the size of the payload), it is necessary to modify two variables in the program: `payloadBufferLength`, which specifies the max size of the payloads, and `payloadLength` which defines the size of the uplink to be sent (of course it must be lower or equal to `payloadBufferLength`)

The payload of the uplinks is organized on 9 bytes, as follows :

Byte 1 et 0	Weight in multiples of 2g grams (the value is 10000 when the measured weight is 20000g). This is allowing to keep the weight on 2 bytes while maintaining a 2g resolution which is more than enough. The max weight is therefore $2 \times 65536 = 131\text{Kg}$, which is also more than enough
Byte 3 et 2	temperature (coded on 2 bytes using function <code>LMIC_f2sf1t16</code> which codes a real between -1 and 1 to a 2 bytes integer

Byte 4	humidity. Coded on 1 byte (resolution 1%)
Byte 5	Atmospheric pressure on 1 byte: The number of mPa greater than 900 is transmitted
Byte 6	Battery voltage on 1 byte : The value x25 is transmitted after removing the 0.
Byte 7	value of the last downlink received by the node
Byte 8	Hive internal temperature: coded on 1 byte: rounded value of actual 10 times (temperature minus 15). If measured temp is 34.6, 196 is transmitted.

The downlink payload is organized on 1 byte, as follows :

Bits 7, 6 (command)	0 0 Trigger of the pump	0 1 Change of measure frequency / standby	1 0 Reset « last downlink » sent with each uplink	1 1 hold measure
Bits 5...Bit 0 (value)	Duration in multiples of 10 seconds (000010 means 40 seconds) Max is therefore 630 seconds)	0 = once per mn 1 = every 2mn 2 = every 5mn 3 = every 10mn 4 = every 15mn 5 = every 30mn 6 = every heures 7 = every 2H 8 = every 6H 9 = every 12H 10 = every 24H 11 = no sleep (*)		Time in multiples of 2mn (max is 126mn)

The "no sleep" mode (command 01-11, i.e. 4B) allows to easily reload a new firmware. Indeed, in normal mode, the processor is most of the time in standby mode, thus unable to manage the serial line that allows to reprogram the microcontroller. When we send the downlink 4B (75 in decimal), the processor never goes into standby and it becomes possible to reprogram it. To make it go back to normal mode (with sleep between two measurements), a downlink redefining the measurement frequency must be sent.

In fact I do not use this function (and I am not sure it works) because I included on the box a switch that allows to cut the power supply of the microcontroller. Knowing that the program starts with a waiting loop, it is possible to reprogram it by sending the new code while it is waiting in this loop (10 seconds). Warning: no 5V supply must be present on the micro USB connector because it goes directly to the microcontroller and cannot be cut by the on/off switch.

Warning : If the pump goes on for a time longer than the interval between two measurements, this will delay the next measurement by the difference between these two times.

The table here below represents the list of possible downlinks :

b7	b6	b5	b4	b3	b2	b1	b0	Décimal	Hexa	Fonction
0	0	x	x	x	x	x	x	**	**	Switch on the pump, xxxxxx is the duration, in multiples of 10s)
0	1	0	0	0	0	0	0	64	40	measureInterval = 1mn
0	1	0	0	0	0	0	1	65	41	measureInterval = 2mn
0	1	0	0	0	0	1	0	66	42	measureInterval = 5mn
0	1	0	0	0	0	1	1	67	43	measureInterval = 10mn
0	1	0	0	0	1	0	0	68	44	measureInterval = 15mn
0	1	0	0	0	1	0	1	69	45	measureInterval = 30mn
0	1	0	0	0	1	1	0	70	46	measureInterval = 1H
0	1	0	0	0	1	1	1	71	47	measureInterval = 2H
0	1	0	0	1	0	0	0	72	48	measureInterval = 6H
0	1	0	0	1	0	0	1	73	49	measureInterval = 12H
0	1	0	0	1	0	1	0	74	4A	measureInterval = 24H
1	0	0	0	0	0	0	0	128	80	Reset last downlink
1	1	x	x	x	x	x	x	*	*	Hold measure (xxxxxx is the duration in multiple of 10mn)

Additionally there is also one (and only one) type of downlink that is on three bytes instead of one byte: this is the command to update the system time. See explanation in [2.1.9](#)

2.1.7 Main Sketch –uplinks and downlinks processing

[LMIC-node](#) code is used as a template (see explanation in the readme on github) in which the user code must be added in three different parts:

- At the beginning of the sketch in order to define the global variables
- In the function `process_work` in order to read the sensor values
- In the function `processDownLink` in order to process downlinks
- At the end of the code, in the `setup()` function in order to initialize the sensors

The different places where user code can be added is clearly tagged with big signs « user code begins here » and « user code ends here » but I had to add code outside of these tags in order to manage the standby mode which is not part of LMIC-node. (See chapter [1.5 Energy](#))

Here is what my code does :

In its first part, it defines the constant that it needs (`#define DHTPIN 10 ;`
`#define DHTTYPE DHT22; #define VBATPIN A7;...`)

It loads the libraries that it needs and defines two global variables of type `uint8_t`:

- `payloadBufferLength` which needs to contain the length of the uplinks in number of bytes ; later in the code, it's needed to also change variable `payloadLength` and give it this exact same value
- `lastDownlink` which is initialized to 0 and which is loaded by the function `processDownlink()` with the value of the last downlink received by the node. All uplinks are sent with a byte (the last one) containing the value of this last downlink so that it's possible to ensure that it has actually been received.

In the second part :

- Preparation of the uplink to be sent : the program reads the sensors (HX711, BME280, DS18B20) measure the battery voltage and put together the payload to be transmitted after coding the different elements
- Processing of the downlinks : the program checks if a downlink has been received and if yes determines which command has been received (bits 6 and 7) and acts accordingly:
 - If those bits are 00, it means it's a "trigger pump" command. The program then reads the value stored in bits 0 to 5, multiplies it by 30 and the result is the number of seconds during which the pump has to be turned ON.
 - If those bits are 01, it's a command to modify the measure frequency according to downlink table in chapter [2.1.6](#). The program will then change variable `doWorkIntervalSeconds` so that it is equal to the number of seconds between two measures, according to this table.

In the last part, which is in reality the beginning of the program, all the sensors are initialized (hx711, bme280 et DS18B20).

2.1.8 Main Sketch – power management

The MCCI_LMIC library works very well but leaves the processor running all the time, which is prohibitive for the battery. Stopping it by putting it in "sleep mode" is not easy because when you wake it up, the LMIC program will have lost its time reference. In fact it counts the ticks to decide when it will send a new uplink. So if we put it to sleep, it will start counting ticks from the moment we wake it up... We had to tinker in the LMIC code outside of the user code areas. See explanations chapter [1.5 Energy](#).

2.1.9 Main Sketch; time management

In "normal" operation, the time is not useful to the program because TTN network manages it: all the measurements received by the network are tagged with the time of reception. There is no need for the microcontroller to know the time and it is therefore initialized to 0 on power-up.

But this notion of time becomes necessary if you want to synchronize the sending of the measurement with the powering up of a gateway to avoid leaving the latter permanently powered up when powered from a battery recharged by a solar panel. A LoRa Gateway is quite power intensive and it's difficult to maintain it always on when sun is not shining (see section [10 Power saving Unit](#))

So I included a mechanism that allows the actual time to be set-up in the microcontroller by sending a downlink of three bytes, containing respectively hours, minutes and seconds. This downlink must contain the precise time of the end of the next measurement.

The format of the time setting downlink is as follows :

- `downlink[2] = seconds`
- `downlink[1] = minutes`
- `downlink[0] = hours`

The program detects that this is a "time update" downlink because it is the only one that includes three bytes (all other downlinks are on a single byte).

In order to send uplinks at well determined times, when the gateway is powered on, the following logic is applied: each time a measurement frequency change downlink is received, the program is put in a waiting loop (before going to sleep mode), until the time of the microcontroller modulo the interval between two measurements is equal to 0. For example if we receive a downlink changing the measurement interval to 15mn, we wait for the time of the microcontroller to be XH00MN, XH15MN, XH30MN or XH45MN before going to sleep.

Because the gateway will be powered on only at certain times, it is necessary that the all microcontrollers are set-up with the same (preferably actual) time, which is not straight forward because the downlink containing the time is processed by the processor when it is sent but rather at the end of the transmission of the next uplink. And we do not know when the next uplink will be sent because when the unit is powered up, it is not synchronized with anything at all. Typically, if the time between two measurements is 15mn, the "reset time" downlink will be processed with a delay of 0 to 15mn, depending on when this downlink is sent.

In order to solve this problem two programs have been developed:

- `getTime.py` which takes as parameters the deviceId of the unit which time needs to be set, the number of minutes between two consecutive measurements on this unit.
Example : `python getTime.py hso-energysaver 15`
- `setTime.py` which takes as parameter the deviceId of the node
(`python setTime.py hso-energysaver`)

The program `getTime.py` will listen to the uplinks sent by the unit which deviceId is given as a parameter and will extract from this uplink its reception time. At this moment we know how long later the next uplink will be processed and therefore at what time (just after) a downlink sent before this next uplink will be processed. The program will therefore add to the reception time of the last uplink the number of minutes between two consecutive measurements (passed as a parameter to the program, together with the deviceId) and place this information in a file `time.dat` in directory `/home/pi/Documents/deviceId`

The program `setTime.py`, which must be launched before the previous one, listens to this directory and as soon as it detects a file in it, reads its content which is the time at which a downlink sent before the reception of the next uplink will be processed. It waits a few seconds then sends the "reset time" downlink command according to the format explained above (downlink of three bytes:

- `downlink[2] = seconds`
- `downlink[1] = minutes`
- `downlink[0] = hours`

Syntax: `python setTime.py hso-energysaver`

2.1.10 Dealing with unexpected hangs thanks to the integrated watch dog

From time to time but very rarely, one driver (I believe this is the BME280) fails to read correctly the sensor data and gets stuck. As a result, the node doesn't communicate anymore, the sleep mode is not activated anymore and the battery gets empty rapidly.

After checking various resources I realized that such things can happen and in order to deal with that, the microcontroller includes a watchdog which can reset the system in case it is not refreshed regularly.

I use the watchdog timer from javos65 available on GitHub [here](#). The examples on GitHub speak for themselves with one exception: the watchdog must be cleared at least once every 32s, which obviously is shorter than the sleep time. I had therefore to disable the watchdog before putting the processor to sleep mode, which is not documented in the provided examples. Looking into the library it's easy to determine that the watchdog can be disabled by calling it with 0 as the main variable: `MyWatchDoggy.setup(0);`

In my code, the watchdog object `MyWatchDoggy` is created in the setup section, then it is enabled just before the sensor reading section (`MyWatchDoggy.setup(WDT_SOFTCYCLE32S);`) and disabled at the end of this section with `MyWatchDoggy.setup(0);`

2.2 JS code for payload decoding (to be loaded on ttn console)

We start from Adafruit example [here](#) which transmits and decodes only two values (temperature and humidity) using 4 bytes. In our case, there are 6 values to be decoded, plus the value of last downlink, which is a total of 9 bytes. It will be done as follows (names that are underlined are the Javascript keys).

- weight : Byte 0 et 1 : weight in multiples of 2g (the value is 10000 for a measured weight of 2000g). Max weight is therefore $2 \times 65536 = 131\text{Kg}$, which is OK.
- temperature : Byte 2 et 3 (coding on 2 bytes using function `LMIC_f2sflt16` which codes a real number between -1 and +1 into a 2 bytes integer).
- humidity : Byte 4 : Coding on 1 byte (resolution 1%, which is OK)
- pressure : Byte 5 : atmospheric pressure on 1 byte: the number de mPa greater than 900
- voltage : Byte 6 : battery voltage on 1 byte : this is the value of the voltage multiplied by 25 with no decimal because the value is on 1 bytes only.
- lastDownlink : this is the value of the last downlink received by the node (allowing to check that it has actually been received)
- internalTemp : hive internal temperature minus 20 and multiplied by 10

Corresponding Javascript code is therefore:

```
function Decoder(bytes, port) {
  var decoded = {};
  codedWeight = bytes[0] + bytes[1] * 256;
  decoded.weight = codedWeight * 2;
  codedTemp = bytes[2] + bytes[3] * 256;
  decoded.temperature = sflt162f(codedTemp) * 100
  decoded.humidity = bytes[4];
  decoded.pressure = bytes[5] + 900;
  decoded.voltage = bytes[6] / 25;
  decoded.lastDownlink = bytes[7];
  decoded.internalTemp = bytes[8] / 10 + 15;
  return decoded;
}

function sflt162f(rawSflt16)
{
  rawSflt16 &= 0xFFFF;
  // special case minus zero:
  if (rawSflt16 == 0x8000)
    return -0.0;
  // extract the sign.
```

```
var sSign = ((rawSflt16 & 0x8000) != 0) ? -1 : 1;
// extract the exponent
var exp1 = (rawSflt16 >> 11) & 0xF;
// extract the "mantissa" (the fractional part)
var mant1 = (rawSflt16 & 0x7FF) / 2048.0;
var f_unscaled = sSign * mant1 * Math.pow(2, exp1 - 15);
return f_unscaled;
}
```

This code can be found in file `ttn-decode.js`

3 INTEGRATION WITH GOOGLE SHEET

This mode of integration is not optimal and I have never managed to have super reliability. Access to Google sheet crashes from time to time (once every 10 or 15 days) and I am not really sure why. I haven't spent a lot of time trying to catch errors in software because the Node-Red/InfluxDB/Grafana integration is preferable in many ways. It is nevertheless very practical to be able to retrieve the data in a Google Sheet file for the calibration of the sensors, in particular to determine the parameters necessary for the temperature compensation of the weight measurement.

3.1 MQTT

3.1.1 Protocol

The integration of TTN with the back-end is done using the MQTT protocol, available on the TTN console. MQTT will make it possible to get all the data received by the TTN server (the uplinks) and also to transmit to this server the data to be sent to the scales (the downlinks)

For explanation on MQTT and Mosquitto : see [this video](#). Summary :

MQTT is a very lightweight and widely used M2M communication protocol over TCP/IP. The sensors correspond to "topics". They communicate with an MQTT server called an "MQTT broker". To receive data from sensors that send data to the MQTT broker, a "subscribe" request is made. To send data to an actuator connected to the MQTT broker, a "publish" request is made. There seems to be 3 connection modes, presenting different connection reliabilities (0, by default, the link is sent once without backup, 1 and 2 are more secure) but the LoRaWAN network only manages the first mode (QoS= 0). This is why I have developed a mechanism to verify that the downlink has been received (the payload of each uplink includes the last downlink received).

The most common broker is "mosquitto" which runs on almost all platforms. It uses the MQTT protocol and therefore makes it possible to send "publish" and to subscribe to "topics".

3.1.2 Mosquitto

The explanations below are for information only because I use a python library to communicate with the MQTT server of TTN.

I nevertheless tested access to the MQTT broker of TTN with the mosquitto commands and summarizes the manipulations made below.

Below is a procedure to load mosquitto on a Raspberry pi but it seems that there is a simpler and safer method consisting of using a so-called "docker" application (see explanations in chapter [5](#))

To load mosquitto on raspberry :

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install mosquitto mosquitto-clients
```

Need to edit the file `mosquitto.conf` in order to check « `include_dir` »:

```
sudo nano /etc/mosquitto/mosquitto.conf
```

It includes the line `include_dir /etc/mosquitto/conf.d`

We now need to create a file `default.conf` containing this directory:

```
sudo nano /etc/mosquitto/conf.d/default.conf
```

in which we put following lines :

allow anonymous false (in order to refuse anonymous connections)

password_file /etc/mosquitto/pwfile

listener 1883 (1883 is the defaults port, which means this line is not really needed...)

We need to create a user with following line:

sudo mosquitto_passwd -c /etc/mosquitto/pwfile userName

The program asks for the user code of userName and store it

Mosquito now needs to be restarted : sudo /etc/init.d/mosquitto restart

3.1.2.1 Subscription

A [page of TTN](#) web site explains the different « topics » to which it's possible to subscribe using the command mosquitto_sub. The syntax is as follows :

```
mosquitto_sub -h hostName -t "topic" -u "userName" -P "password" -d
(option -d means "enable debug messages », userName is the application_id of TTN where @ttn is added at the end)
```

- Hostname is [eul.cloud.thethings.network](#)
- password is the API key provided by TTN (on the integration page)
- user name is {applicationId}@{tenantId} knowing that tenantId is the host name (ttn). See « [MQTT integration](#) » page
- The different « topic » are listed here below and it's possible to subscribe to all topics by replacing "device_id" by "#"

- v3/{application id}@{tenant id}/devices/{device id}/join
- v3/{application id}@{tenant id}/devices/{device id}/up
- v3/{application id}@{tenant id}/devices/{device id}/down/queued
- v3/{application id}@{tenant id}/devices/{device id}/down/sent
- v3/{application id}@{tenant id}/devices/{device id}/down/ack
- v3/{application id}@{tenant id}/devices/{device id}/down/nack
- v3/{application id}@{tenant id}/devices/{device id}/down/failed
- v3/{application id}@{tenant id}/devices/{device id}/service/data
- v3/{application id}@{tenant id}/devices/{device id}/location/solved

In my case here is the command:

```
mosquitto_sub -h eul.cloud.thethings.network -t "#" -u "ardbeescale-af01@ttn" -P
"NNSXS.E5LAS7FKQAFUHNFXIDRKQOBAAFSCLGQPZIJJE5SI.3C4MRNGCMAMZ6UE6VFJ3A653VPH2FICMF6UNNCYYKIHN5YVGVMF"
-d
```

(**warning** : the double quotes are not the same under word and under textEdit... The quotes to be used are those available in Terminal application)

3.1.2.2 Publications

Syntax is as follows :

```
mosquitto_pub -h thethings.example.com \
-t "v3/app1@tenant1/devices/dev1/down/push" \
-u "app1@tenant1" -P "NNSXS.VEEBURF3KR77ZR.." \
-m '{"downlinks":[{"f_port": 15,"frm_payload":"vu8=","priority":
"NORMAL"}]}' \
-d`
```

In my case, in order to send 0C (which is wA== base 64) :

```
mosquitto_pub -h eul.cloud.thethings.network -t "v3/ardbeescale-af01@ttn/devices/hive01-
test2af/down/push" -u "ardbeescale-af01@ttn" -P
"NNSXS.E5LAS7FKQAFUHNFXIDRKQOBAAFSCLGQPZIJJE5SI.3C4MRNGCMAMZ6UE6VFJ3A653VPH2FICMF6UNNCYYKIHN5YVGVMF"
-m '{"downlinks":[{"f_port": 15,"frm_payload":"wA==","priority": "NORMAL"}]}' -d
```

It's possible to replace /push by /replace in order to clear the downlinks present in the queue.

It's also possible to ask for an AKN. See TTN doc [here](#).

3.1.3 MQTT & python

3.1.3.1 Principle

See explanation about how to use this library [here](#) and there is also an explanatory video [here](#).

The python library used to retrieve data from an MQTT broker and publish data is called paho. Documentation is available [here](#).

It needs to be installed on raspberry pi:

```
pip3 install paho.mqtt
```

The main class of this library is `Client`. It includes several methods :

- o `connect()` and `disconnect()`
- o `subscribe()` and `unsubscribe()`
- o `publish()`
- o `username_pw_set()`

The different steps are as follows:

Import in a python program: `import paho.mqtt.client as mqtt`

Instantiation of the client: `client =mqtt.Client(client_name)`

`client_name` is optional. If left blank the system will affect a name

Connection to broker : `connect(host, port=1883, keepalive=60, bind_address="")`

with TTN, host is : `eul.cloud.thethings.network`

In all the examples that I found, the connection to the broker is done by the command above, without specifying the user_name or the password. But by reading the docs of the library, we see that there is a command (method) to specify these parameters with the connection:

```
username_pw_set(username, password)
```

Therefore, in order to publish a message, the sequence is as follows :

```
USER = "ardbeescale-af01@ttn"
PASSWORD = "NNSXS.E5LAS7FKQAFUHNFXIDRKQOBAAFSCLGQPZIJES5I.3C4MRNGCMAM26UE6VFJ3A653VPH2FICMF6UNNCYYKIHNSYVGVMFPA"
DEVICE_ID = "hive01-test2af"
PUBLIC_ADDRESS = "eul.cloud.thethings.network"
PUBLIC_ADDRESS_PORT = 1883
mqttClient = mqtt.Client()
mqttClient.username_pw_set(USER, PASSWORD)
mqttClient.connect(PUBLIC_ADDRESS, PUBLIC_ADDRESS_PORT, 60)
topic = "v3/" + USER + "/devices/" + DEVICE_ID + "/down/push"
hexadecimal_payload = '0C'
fport = 100
# Convert hexadecimal payload to base64
b64 = b64encode(bytes.fromhex(hexadecimal_payload)).decode()
msg = '{"downlinks":[{"f_port":"' + str(fport) + '","frm_payload":"' + b64 + '","priority": "NORMAL"}]}'
result = mqttClient.publish(topic, msg, QOS)
```

It's possible to replace `/push` by `/replace` in order to clear the other downlinks present in the queue. It's possible to send several downlinks (a downlink is an array)

It's also possible to ask for an AKN. See documentation [here](#).

For "subscriptions" it's needed to use the so called « callback functions ». Basically, each time a message is received (or a connection is established, or broken...), the script launches a function that will process this event. These are called "callback functions". For each of those events, you have to

define what the corresponding callback function does and tell the instantiation object of the class (the MQTT client) what this function is. And for the callback functions to be activated, the program must listen; this is what we do with the method `loop(timeout)`, `timeout` being the time during which this function blocks the program. It's possible to use `loop_forever()` in which case the program will loop forever as the name suggests or to use `loop_start()` to launch an independent thread which do not block the program, which can be stopped with `loop_stop()`

Callback functions cannot return a value. There are several possibilities to retrieve the values of a received message:

- Declare a global variable using the global statement
- Use a list declared in the main function.
- Use a dictionary declared in the main function.
- Use an Object declared in the main function.

3.2 Python code running on Raspi serveur

In order to understand paho.mqtt, see explanation [here](#).

This is the code that retrieves the uplinks and the one that places them in a gsheets file.

3.2.1 Programs & data organization

Initially the programs were written to manage multiple scales (6) by loading data in the correct order into a google sheet file so that a dashboard is updated correctly. Today I use these programs only to calibrate scales (determination of temperature compensation parameters) and therefore some parts of the code may look weird because I kept the initial code structure which contains complexities that have become unnecessary .

Each hive sends its own uplinks. All the hives of the same apiary are on the same application in TTN so the paho mqtt library will receive the uplinks of all the hives of the same apiary (you can choose to subscribe to all the topics or to only one) .

The `application_id` that I have chosen is `ardbeescale-af01` and the node names are respectively : `hso-scale01` to `hso-scale06` plus `hso-energysaver`.

Two pythons programs are used:

getdataFromUplink.py

Retrieves the uplinks from the MQTT server and store them in a file (the callback function which receives the messages will place the data received in the directory `application_id` of directory `hiveDataRootDir` (`hiveDataRootDir` being defined in the configuration file). New files created in this directory have the form `YYMMDDTHMM#deviceId`; they contain a python list : `[weight, temperature, humidity, pressure, voltage]`

(internal temperature measurement has been added later and is therefore not taken in account in this "google sheet" integration)

More details on this program in [3.2.3](#)

storedata.py

Processes the file created by `getdataFromUplink.py` and store corresponding information in a google sheet file. Each time a file is created in the target directory, the program will catch it thanks to the `pyinotify` logic and will store corresponding data in the google sheet program. The way the data are stored in the gsheets file may seem weird; this is again because originally the file was used in a different purpose (example: the data are stored at the bottom of the file and rolled up by one row each time there is a new measure).

More details on this program in [3.2.4](#)

Note:

I had to install `gsread` by doing a git clone (from the `pythonScripts` directory of my raspberry) (`git clone https://github.com/burnash/gspread.git`) and for the set-up, the command is `sudo python3 setup.py install` to be launched from within directory `gsread`

It's also needed to install module `oauth2client` :

```
sudo apt-get update
sudo apt-get install python3-oauth2client
```

3.2.2 Python configuration file

Its name is `config_ardbeescale-af01.py` and it contains following parameters:

- `hiveDataRootDir` : root directory in which `getdatafromUplink.py` will store the data received from the MQTT server.
- The list of nodes for this application :
`hiveList=[device_id1, deviceId2, ,,deviceId6]` The program checks if the uplink it get from the MQTT broker corresponds to a node that it needs to register in the gsheets file. The position of the `deviceId` in the list tells the program in which column of the google sheet file it needs to store the data.
- `hiveSelectedForTHP` : It is the number (1 to 6) of the scale which temperature, humidity and pressure are stored in the google sheet. The others, which also measure external T, H and P are not stored in the file.
- `measureIntervalMn` : it's the time, in number of minutes, between two consecutive measures. The program needs to know that because it receives uplinks at different times and group the uplinks on a same line in the google sheet and it needs therefore to calculate round times.
- `gSheetLastLineNbST` : for some reasons that are not valid anymore now that I am not using any more the gsheets file to create a dashboard, new lines are added at the bottom of the gsheets file. Each time a new line has to be added, the first line (after the title line) is deleted and the new data are added line `gSheetLastLineNbST`
- `gSheetLastLineNbLT` : the sheet « LT » stores the first data of each day. Same principle : a new line is added at the bottom of the file.

3.2.3 Retrieving data from MQTT and temporary storage

Script `getdataFromUplink.py`

The script retrieves all the uplinks issued by the nodes listed in the configuration file `config_ardbeescale-af01.py` in a python list named "listOfScales" corresponding to the user (=application) mentioned in the program together with the credentials (which can be retrieved from TTN console in the "MQTT" integration page).

The program from `mqtt paho` library is launched by the line `mqttc.loop(10)` - with 10 meaning a timeout of 10s or by the line `mqttc.loop_forever()`. Each time an event occurs (a connection, a subscription, an incoming message...), a « callback function » is called (`on_connect`, `on_subscribe`, `on_message`...); each of those callback functions must be defined in the program:

```
mymqtt = mqtt.Client()
mymqtt.on_connect = on_connect
mymqtt.on_subscribe = on_subscribe
mymqtt.on_message = on_message
mymqtt.on_disconnect = on_disconnect
```

The callback functions are called by the occurrence of an event coming with several parameters. If the event is `on_message`, one of those parameters is `message` and it contains all the information pertaining to the uplinks just received (`message.payload`, `message.topic`, `message.qos`...).

Therefore this is in the callback function `on_message` that we will retrieve the information contained in the uplink; they can be found in the json object `message.payload`.

It is therefore needed to load the python module « json » which includes a method called `loads` allowing to get the uplink content in a python dictionary :

```
import json
# parse jsonObject:
jsonDict = json.loads(message.payload) # the result is a Python dictionary:
```

In order to understand what this dictionary is containing, the best is look at the structure of the json object sent by TTN (`message.payload`). This is available on TTN web site [here](#). In summary the most important keys of the dictionary are:

"end_device_ids" (which is also a dictionary, containing device_id, dev-EUI, etc...)
"received_at" (date and time the message was received in format ISO 8601 UTC)
"uplink_message" (dictionary containing all the uplink info)

This dictionary `uplink_message` contains a key `frm_payload` which is the raw (=not decoded) payload and a key `decoded_payload` allowing to access to a dictionary which keys are field names of the values transmitted (temperature, humidity...) as defined in the JavaScript file decoding the data on TTN, the values of those keys being those transmitted by the node.

Retrieving device_id:

If `jsonDict` is the dictionary coming from the json object `message.payload`, `device_id` can be retrieved as follows:

```
endDevice_idsDict = jsonDict["end_device_ids"]
device_id = endDevice_idsDict["device_id"]
```

Retrieving of uplink receiving time :

```
received_at = jsonDict["received_at"]
```

Warning : this is UTC time, shifted by 1H versus CET in winter and 2H in summer.

Retrieving of uplink data :

```
uplink_message = jsonDict["uplink_message"]
uplink_message is a dictionary containing decoded_upload :
```

```
decoded_payload = jsonDict["uplink_message"]
```

The data sent by the node can be found in the dictionary `decoded_payload`

```
weight = decoded_payload["weight"]
temperature = decoded_payload["temperature"]
humidity = decoded_payload["humidity"]
pressure = decoded_payload["pressure"]
voltage = decoded_payload["voltage"]
```

These data are then loaded in a list `[weight, temperature, humidity, pressure, voltage]` which is stored in a file in a directory that is hard coded in the program line 63. File name includes device_id and time of receiving :

```
/home/pi/Documents/hiveData/ardbeescale-af01/2021-11-18T21:41#hso-cale01
```

3.2.4 Data storage in a google sheet file

Script `storedata.py`

Warning: as opposed to `getdataFromUplink.py` which stores the data in a directory that is hard coded, the directory in which `storeData.py` detects file creation is not hard coded but specified in the configuration file `ardbeescale-af01.py` this is because I do not use this integration for anything else than calibration I have not taken the time align the two programs. It is therefore important to check that the variable « `hiveDataRootDir` » in the configuration file is the same as the hard coded directory of `getdataFromUplink.py` line 63.

storeData.py also kept the mechanics of setting the name of the configuration file (initially there was one configuration file per apiary because the gsheets integration system was made to manage several apiaries). It is now useless but it is not changed.

Each time a file is created in /home/pi/Documents/hiveData/ardbeescale-af01/ pyinotify will detect it and will store it in the gsheets file (which name is simply the application name, ardbeescale-af01 in my case)

There are several possible implementations of pyinotify. I have used what I did in my old rasbeescale project with following code :

```
import pyinotify
wm = pyinotify.WatchManager() # Watch Manager
# events to watch:
mask = pyinotify.IN_DELETE | pyinotify.IN_CREATE | pyinotify.IN_CLOSE_WRITE
class EventHandler(pyinotify.ProcessEvent):
    def process_IN_CLOSE_WRITE(self, event):
        print("File just closed:", event.pathname)
    def process_IN_CREATE(self, event):
        print("File created:", event.pathname)
handler = EventHandler()
notifier = pyinotify.Notifier(wm, handler)
wdd = wm.add_watch("/home/pi/Documents/pythonScripts", mask, rec=True)
notifier.loop()
```

The program retrieves the device_id and the time of uplink reception from the file content and the data of the uplink from the list stored in this file. The complete path of the file is returned by pyinotify and has following form:

```
path='/home/pi/Documents/hiveData/ardbeescale-af01/2021-11-18T17:03#hive01-test2af'
```

The formula to get device_id is : device_id=path[path.index("#")+1:]

(string[n:p] returns the characters n to p of string - if p is omitted, it returns all the characters from n up to the end of the string.

To get the date of reception of the uplink :

```
year = path[path.index("T")-10: path.index("T")-10+4]
month = path[path.index("T")-5: path.index("T")-5+2]
day = path[path.index("T")-2: path.index("T")-2+2]
hour = path[path.index("T")+1: path.index("T")+1+2]
mn = path[path.index("T")+4: path.index("T")+4+2]
```

All the hives do not send their data at the same time, but we want to store in the gsheets file all the data received "about the same time" on the same line. With each uplink it is therefore necessary to determine whether to add a line or whether it is the transmission of a hive which is "almost" simultaneous with an uplink already received. For this, we will calculate the arrival time of the uplink in number of minutes since midnight, modulo the interval between two measurements (found in the measureIntervalMn variable, the same for all the hives). Thus, if the interval between two measurements is 5 minutes, two uplinks received respectively at 4:02 p.m. and 4:03 p.m. will be stored in the gsheets file at 4:00 p.m. When receiving each uplink, we calculate this modulo the interval between two measurements then we will read the reception time of the last uplink received on the last line of the gsheets file, also modulo measureIntervalMn. If these two values are the same, we will simply store the data of the uplink on the last existing line of the gsheets file whereas if the reception time of the uplink modulo the time interval between two measures is later than the value of the time read on the last line of the gsheets file (always modulo the same number), we will add a new line in this file

Retrieval of the time of the last uplink from the google sheet file (the result is a string):
(see chapter [8 below](#) on how to install the python google sheet library)

```
lastLineDateTime = wks.cell(1,config.gSheetLastLineNb).value qui est de la forme
'20/11/2021 21:42:51'
```

How to get the time of the last line:

```
lastLineTime = lastLineDateTime[lastLineDateTime.index(" ") + 1 : lastLineDateTime.index(" ") + 6]
```

Then the number of minutes since midnight :

```
lastLineMn = int(lastLineTime[0:2]) * 60 + int(lastLineTime[3:5])
```

And finally this same number, minus the value modulo the measurement interval:

```
lastLineMnRounded = lastLineTime - lastLineMn % config.measureIntervalMn
```

The same calculation is done on the reception time of the uplink, which was calculated before from the file containing the uplink:

```
lastUplinkMn = 60 * int(hour) + int(mn)
```

```
lastUplinkMnRounded = lastUplinkMn - lastUplinkMn % config.measureIntervalMn
```

And to determine if we are on a new line or if it is the measurement supposed to be "roughly" concomitant with those already received, we compare `lastLineMnRounded` and `lastUplinkMnRounded`.

If the two values are different, we will access the gsheet file to delete the top line and add a new line at the bottom (we proceed in this way so that the graphs on the dashboard continue to point to the right data as we fill in the data. When adding a line, we check whether there has been a date change since the last measurement and if so, we copy the line of data corresponding to the last measurement of the previous day in the "LT" sheet (for long term).

Then we simply load the data corresponding to the uplink being processed in the right cell of the gsheet file. Detecting the number of the hive to which the uplink being processed corresponds (by accessing the configuration file) makes it possible to put the data in the right cells (column). Moreover, if it is the hive that we want to use to record temperature, humidity and pressure (this hive number is specified with the parameter `hiveSelectedForTHM` of the configuration file) we also load this data on the line. The temperature, humidity and pressure data from the other sensors are ignored.

The date requires a special treatment because it is available in the script in the following variables: `year`, `month`, `day`, which represent the date of reception of the uplink and for the hours and minutes in the variable `lastUplinkMnRounded` which represents the number of minutes since midnight. We convert the latter into hour and mn and we convert these 5 variables into a string to have a DD/MM/YYYY HH:MM type format that we put in a `dateTimeString` variable. With this character string, we will create a datetime type object:

```
dateTimeObj = datetime.datetime.strptime(dateTimeString, '%d/%m/%Y %H:%M')
```

It remains to convert this object into an excel date:

```
delta = dateTimeObj - datetime.datetime(1899, 12, 30)
```

```
XlDate = float(delta.days) + (float(delta.seconds) / 86400)
```

Accessing google APIs to update gsheet file is simple (see explanation chapter [8](#)) but I added a mechanism to manage cases where we exceeded the maximum number of API calls per unit of time defined by google (I think it's 100 every 100 seconds). This mechanism makes it possible to slow down the accesses to go below this limit. With a frequency of uplinks of 4 or 5 per hour, this mechanism is useless and can be removed.

3.2.5 Installation

The google sheet file name used to be a parameter (when I wanted to use this type of integration in real life) but now it is hard coded in two python script files `getDataFromUplink.py` and `storedata.py`

It is necessary that the gsheet file be filled with at least two lines of measures otherwise the program crashes because it tests if the day of the last measurement is the same as the day of the previous measurement (the change of day triggers the registration of the last measurement in the "sheet" named LT. There is no error catching.

There are a few libraries to load (remember to update and upgrade before...):

- `pyinotify` (`sudo pip install pyinotify`)
- `paho.mqtt` (`sudo pip3 install paho.mqtt`)

The programs to load are:

- `getdataFromUplink.py` This is the program that listens to the TTN broker, which decodes the data received and stores it in the corresponding apiary directory (`/home/pi/Documents/hiveData/application_id`). The file contains a python list and the file name is in the form of
`filePath = "/home/pi/Documents/hiveData/"+application_id+"/"+received_at[0:16]+"-"+device_id`
- `storedata.py` This is the program that takes care of storing the data received in the gsheet file (specified in the program), each time a new file is dropped in the directory `/home/pi/Documents/hiveData/application_id` by the script `getdataFromUplink.py`

My scripts are located in the directory `/home/pi/Documents/pythonScripts`.

The parameter `hiveDataRootDir` must be defined in `config_application_id.sys` (in my case `hiveDataRootDir = "/home/pi/Documents/hiveData"`)

To generate the google sheet file, the best is to copy an existing one and renaming it. You must then follow the procedure explained in chapter [8](#) in order to install the right library which will access the gsheet (gsheet).

The name of the credential file must be `application_id_secret.json` (in my case `ardbeescale-af01_secret.json`)

Need also to create the directory containing the log files (which name is specified in parameter `logFile` in config file)

4 BLYNK

4.1 General Principle

To communicate with the scales, I use the Blynk IOT technology which allows, from a smartphone app, to send commands to (or read sensors from) IOT devices. In our case, Blynk will be used to communicate with only one device: the Raspberry pi which receives commands from the Blynk application from which it will determine which downlinks to send to which scale. Communication in the other direction will be used to retrieve the "lastDownlink" value from the uplinks sent by each scale together with the time at which it was received. This is done to verify that the last downlink sent has actually been received. The other data (weight, temperature, etc...) are visualized with Grafana thanks to the integration described in the next chapter: [5 node-RED - influxDB - Grafana](#). Blynk app is not used for that purpose (it could) except external temperature, humidity and atmospheric pressure which are also retrieved from the raspberry pi receiving the uplinks, and shown on the Blynk app screen.

In the future, it is possible to consider using Blynk to visualize the data of the different scales. To do this, you simply subscribe to the corresponding topics, decode them and send to Blynk the value of the different elements.

Blynk's billing method is based on the number of IOT devices controlled by the application. In our case there is only one, so the application is free.

4.2 Blynk set-up

Need to go to blynk.io and log in. A « Template » needs to be created. I understand that a template is a group of widget applicable to all devices of the same kind. The name of the template must be entered, as well as the hardware type and the type of connection.

The site gives then what is called « tokens » :


```
#define BLYNK_TEMPLATE_ID "TMPL1dONedlg"  
#define BLYNK_DEVICE_NAME "ardbeescaleHSO"
```

Then we need to add a "datastream" (which are the old "Virtual Pin") corresponding to the data we want to manipulate. In the free version you can define a maximum of 30 widgets per template.

On the smartphone app (or in the web dashboard), you define the widgets (a button to trigger the pumps, a field to display data...) and you associate to each of these widgets a virtual pin. In the first Blynk docs, there was a notion of "read" and "write" to respectively transmit a data to the smartphone (display) or from the smartphone (command) but it seems that these notions of read and write have disappeared in version 2 because each time, the syntax of the code is "write".

The distribution of the virtual pins chosen is as follows:

	Déclenchement pompe (write)	Reset last downlink (write)	Last downlink (read)	Time of last uplink (read))
Pompe #1	V1	V7	V13	V18
Pompe #2	V2	V8	V14	V20
Pompe #3	V3	V9	V15	V21
Pompe #4	V4	V10	V16	V22
Pompe #5	V5	V11	V17	V23
Pompe #6	V6	V12	V18	V24

V0 : 12V battery voltage

V25 : External temperature (mean temperature of all scales)

V26 : External humidity (mean humidity of all scales)

V27 : External pressure (mean pressure of all scales)

V28 : choice of the number of hours during which the measurements are blocked (selecting 2 means you can manipulate the hive during two hours w/o polluting the weight chart because no measurement will be taken during 2 hours)

V29 : Triggering the blocking of measures. Once V28 has been set, the measures start to be blocked when the V29 button is pushed

Another Blynk screen could be develop to add some functionalities which currently can be accessed only from a python program running on the raspberry pi. Those functionalities are:

- Selection of measurement frequency
- Triggering of measurement frequency change
- Number of minutes that the gateway is powered-up before the time of sending the uplink
- Number of minutes during which the gateway is powered after the time of sending the uplink
- Permanent power supply to the gateway (C0)
- Intermittent power supply to the gateway (C1)
- Setting the time of a scale

4.3 Programming

To write a data on the smartphone, via the virtual pin V2 the code is the following:

```
import BlynkLib
import time

BLYNK_AUTH = 'H9m3Xn0pY9SvtZzy01RtL1MKiw3qZKoy'
# initialize Blynk
blynk = BlynkLib.Blynk(BLYNK_AUTH)

n=0
while True:
    blynk.run()

    print("1 sec elapsed, sending data to the server...",n)
    blynk.virtual_write(2, n)
    n+=1
    time.sleep(1)
```

A number incrementing by one unit every second will be sent to the smartphone in a widget that is able to display this number

In the other way, the code to send a data from the app to the raspberry pi is as follow (the data is entered using an ad hoc widget connected to the right virtual pin, V0 in the example here below):

```
import BlynkLib

BLYNK_AUTH = 'H9m3Xn0pY9SvtZzy01RtL1MKiw3qZKoy'

# Initialize Blynk
blynk = BlynkLib.Blynk(BLYNK_AUTH)

# Register virtual pin handler
@blynk.on("V0")
def v0_write_handler(value):
    print("Tigger pump1; l'app vient d'envoyer : ",value[0])

while True:
    blynk.run()
```

In order to read the contents of the uplinks emitted by the scales, we use the MQTT python library which, like Blynk, includes an infinite loop. It is therefore impossible to make a single program collecting the data from the uplink and displaying them on the smartphone.

Two programs are needed :

- `getStoreUplinks.py` uses the MQTT library to detect the arrival of uplinks (one per scale) then creates a file containing the data (payload) of this uplink and finally places it in a directory defined in the configuration file `ardConf.py`. The file name in which the payload is stored is defined in the configuration file (`lastUplink.dat` is the default). This file contains a list which all the payload elements:
`[uplinkTime,lastDownlink, weight, temperature,humidity,pressure,voltage,internalTemp]`. It is stored in a directory which is `/rootDirectory/application_id/device_id` (`rootDirectory` is a parameter defined in `ardConf.py` - `/media/pi/PI4HOME_HDD/uplinks/` in my case). Knowing that Blynk will regularly (once every 10s) read the stored data, it is important not to put this directory on the SD card of the pi. So I store the uplinks on an external disk. Be careful to change the access rights so that the program can access them:
`sudo chown -R pi /media/pi/PI4HOME_MAIN`
- `blynkMain.py` is in charge of writing this data on the smartphone. It will read the files in which the previous program is storing the uplinks payloads and display the corresponding information on the smartphone. In addition, this same program manages the buttons of the Blynk application of the smartphone allowing to trigger the pumps or to reset the value of the last received downlink.

For triggering the pumps, I have decided for simplicity that the volume of syrup is not adjustable in the app: the same quantity is always injected ; however it can be changed in the python configuration file `ardConf.py` (each scale can have a different volume).

Here is what the configuration file `ardConf.py` contains :

- All TTN access credentials

- The list of device_ids whose uplinks the program will listen to, in two distinct lists : `listOfScales[]` et `otherDevices[]`. Indeed, since these two categories of devices send uplinks of different formats, the program treats them differently.
- File name in which the uplinks will be stored
- Directory in which those files will be stored

The configuration file also contains a mapping used by the program sending downlinks (`sendDownlink(DL)` function, part `blynkFunctions.py` library). In order to send downlinks, the program `blynkMain.py`, uses this function `sendDownlink(DL)` where `DL` is a string parameter. The configuration file `ardConf.py` contains a dictionary which keys are these `DL` strings and which values are lists containing 2 elements: the first one is the downlink to be sent and the second one is the device numbers to which the downlink must be sent (this is the index+1 of the `listOfScales[]` list).

4.4 Blynk install on raspberry

Python libraries and corresponding documentation can be found on :

<https://github.com/vshymanskyi/blynk-library-python>

```
pip install can be used to install the library:  
pip install blynk-library-python
```

But it seems that this command installs the old version of the library because I had an error message "invalid token" characteristic of a version miss-match. I had to go to the directory where the library was installed (`/home/pi/.local/lib/python3.9/site-packages`) and change the content of the `BlynkLib.py` file by the one present on github repository.

Alternatively, it's possible to manually load the library :

From within the directory in which the library must be installed, type:

```
git clone https://github.com/vshymanskyi/blynk-library-python.git
```

The `pythonpath` must be updated with the directory in which this library has been installed :

```
sys.path.append('home/pi/Documents/pythonScripts/blynk-python-library')
```

4.5 Configuration of Blynk app

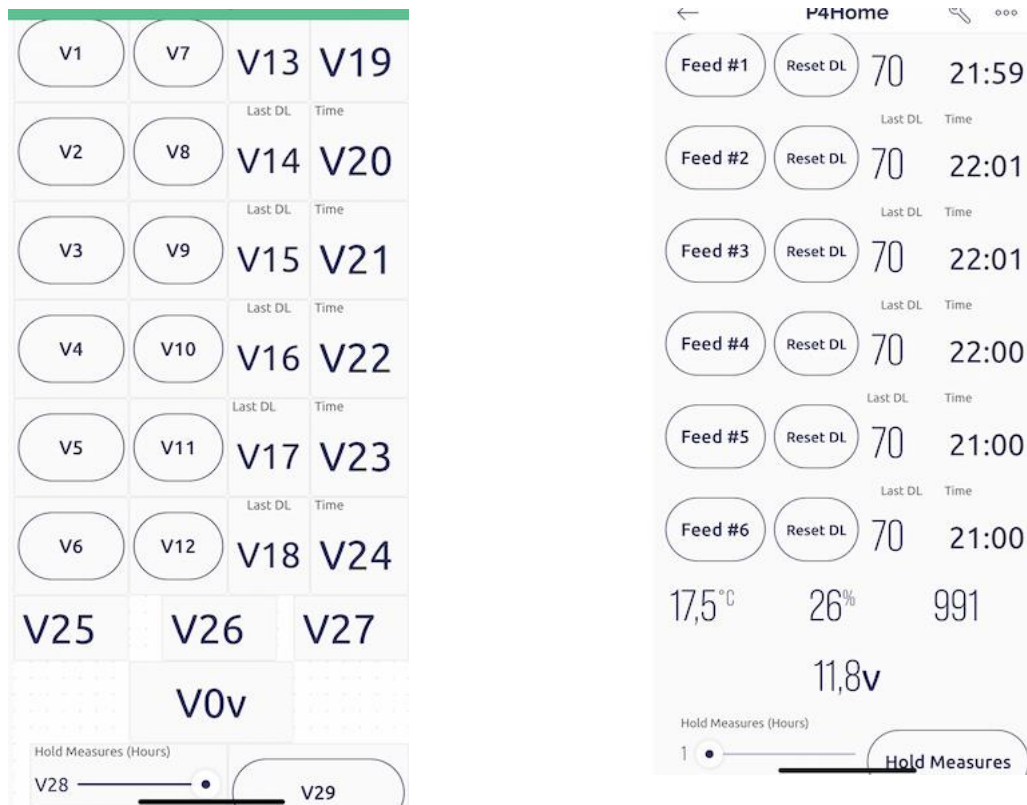
I am using only the free widgets available in Blynk (a lot of widgets used to be free in the previous Blynk version but only a few of them, the least fancy ones, are still free in the new version). A virtual pin has to be associated with each widget. My configuration is as follows:

V1 to V12 are virtual pins associated to "output" widgets (sending a command to the devices from the app). Those widgets are basic buttons: V1 to V6 are buttons triggering the sirup pumps et V7 to V12 are buttons resetting the value of "lastDownlink".

V28 is also connected to an "output" widget which sends to the raspberry python program the time during which no measure should be taken (during hive manipulation). The time during which no measure are taken start when V29 button is pressed.

V0 and V13 to V27 are connected to "input" widget, which show on the screen values of parameters collected on the devices: V0 is the value of the 12V battery, V13 to V18 are showing the value of the last downlink received by each device, V19 to V24 are showing the time at which the last downlink has been received, V25 to V27 are respectively the temperature, humidity and atmospheric pressure measured in the apiary. Knowing that these value are measured by each scale, the values showed on the app are the average of all scales that are considered "valid". This allows to eliminate one scale in case a sensor doesn't work. Selecting the scales which temperature, humidity and pressure measurements are valid is done in the `ardConf.py` configuration file

Below are screen shots of the Blynk app. On the left the configuration screen and on the right the actual user screen:



4.6 Update as of February 2023

Blynk has released a note stating that as of February 28th 2023, the free version will accept only 10 virtual pin per template as opposed to 30 so far. I initially considered changing the set-up explained above in order to accommodate the new limitation by creating several templates instead of only one but it doesn't seem to be possible and the only way seems to be an upgrade to the pay version (4.99€/month with yearly payment).

5 NODE-RED – INFLUXDB – GRAFANA INTEGRATION

5.1 Principle

This is an alternative to my original way of doing things described above (retrieving data from TTS/TTN with a python program (MQTT paho library) that will store it on a google sheets file).

Here, we use node-RED to interface with TTN's MQTT broker and retrieve the uplinks sent by the nodes. Still using node-RED, we connect to a "time series" database called influxDB in order to properly store those uplinks and then we use the Grafana reporting tool to view the data.

Downlinks can be sent using the dashboard functions of node-red, also accessible from a smartphone but I haven't implemented this (yet). Instead, the few most important downlinks are sent using Blynk app and for all the others, I am sending them using a python program running on a raspberry pi.

For this integration using Node-Red, InfluxDB and Grafana, you just need a small server like raspberry pi for example with the three applications installed. In some Linux distributions it seems that node-red is loaded by default. Otherwise, to install new applications, it seems that a modern way of proceeding is to use what is called "containers" which isolates the applications from the OS thanks to a layer called "docker". There is one container per application.

About docker, see this [video](#) from Andreas Spiess. Summary :

Unlike virtual machines that include the OS, containers are independent of the OS and do not encapsulate it. They are therefore much lighter:



Docker is the de facto standard for managing containers.

By going to hub.docker.com, you can load the containers you need, but there is a simpler way: using "docker-compose". This is a utility that will load all the containers that have been specified in the "docker-compose.yml" file and that allows to place the data outside the containers in order to save them easily. To create this file, we use the utility developed by Graham Garner which was initially called IOTstack but which has been migrated and is now found at <https://github.com/SensorsIot/IOTstack> . The command to load this utility is the following :

```
git clone https://github.com/SensorsIot/IOTstack.git ~/IOTstack
```

Need to read [getting started](#) mentioned in the readme :

Summary :

See https://sensorsiot.github.io/IOTstack/Basic_setup/

In the /IOTstack directory, menu.sh will launch the IOTstack application allowing to create the docker-compose.yml file containing the information relative to the containers we want to install. In order to launch it, type ./menu.sh from within the IOTstack directory. The program includes an option to create the stack directly from within the program. Alternatively, it's possible to exit the program and type "docker-compose up -d" to install the stack. It's possible (and advisable) to change the parameters of docker-compose.yml so that the data files of each container are located in the place you want (on a HDD for example). This allows you to delete a container without deleting the data and to easily backup the data and not the whole container. In order to do so, the "volume" section of docker-compose.yml must be changed. For example, in the case of grafana, the initial file includes following lines:

```
volume:
- grafana-data:/var/Lib/grafana
```

They need to be replaced by:

```
volume:
- /directoryWhereYouWantYourData:/var/Lib/grafana
```

For a reason that I fail to understand, this initially didn't work because of permission issues and I had to change owner and permissions on the hard drive on which I wanted to store my data.

It is strongly advised to learn the key commands of docker, which are:

- **docker restart container_name:** to restart a container
- **docker volume prune** remove all the volumes of unused containers
- **docker system prune** remove all the images of unused containers
- When a change is made in the stack (such as an addition of a container), all the containers must first be uninstalled with the command **docker-compose down** and everything must be then reinstalled with **docker-compose up -d**
- To remove a container: it must first be stopped (from portainer) and then removed with the command **docker-compose rm**

- Command to show all the running containers: **docker ps**
- **docker-compose up** processes the file `docker-compose.yml`. If the containers specified in this file already exist, they will be stopped and re-created (volume specifications will be preserved). In order to avoid that: **docker-compose up --no-recreate**
- To load only one container: **docker-compose up -d «container»**
- To stop only one container: **docker-compose stop «container»**
- **docker-compose stop**: stop running containers w/o removing them
- **docker-compose down**: Stops containers and removes containers, networks, volumes, and images created by up
- To remove only one container: **docker-compose rm --force --stop -v «container»**
- **docker images** list all the available images on the current machine (see hub.docker.com for a list of all existing containers)

Andreas Spiess strongly suggests to install portainer (portainer.io) to easily manage all installed containers.

Once everything is installed, a browser must be open to access the different applications :

- node-RED on port 1880, by typing the address `localhost:1880`
- portainer on port 9000,
- Grafana on port 3000.

influxDB has no web interface: access is through a terminal which is inside container. The command is :

```
docker exe -it influxdb /bin/bash
```

We then have a prompt which is the internal terminal of the container on which we must type `influx` in order to access to all the commands of the database.

Alternatively, you can also access the container's internal terminal through portainer by clicking on the > icon.

IOWatch can also allow to forbid too frequent swaps of the RAM ("disable swap" option in `menu.sh`) and also to install "lock2ram" to decrease the frequency of accesses to the SD card.

To update the containers from time to time :

Updating the images

If a new version of a container image is available on docker hub it can be updated by a pull command.

Use the `docker-compose down` command to stop the stack

Pull the latest version from docker hub with one of the following command

```
docker-compose pull or the script ./scripts/update.sh
```

There is a [second video](#) from the same author, very interesting because it summarizes the first one and explains the integration of an MQTT sensor with node-RED, influxDB and Grafana.

5.2 Node-RED

node-RED explanation video: [here](#) (First of a series of 3, in French)

Or [this one](#) which is quicker but gives a good idea of what can be done.

Node-RED is a kind of IOT programming language using pre-programmed "blocks" that are linked together and that can be customized in `node.js`

Node-RED allows the integration of many applications and data sources, local or remote (sensors, tweets, amazon web services...) and can be mounted on a local or remote server, or on a standard computer.

Demo of an integration example of MQTT – NodeRed – InfluxDB – Grafana : see [this vidéo](#) (from a guy dealing with air quality sensors).

There is also a [video](#) on The Thing Stack explaining how to integrate TTN, InfluxDB and telegraf. I have not used it and have preferred to go with node-red.

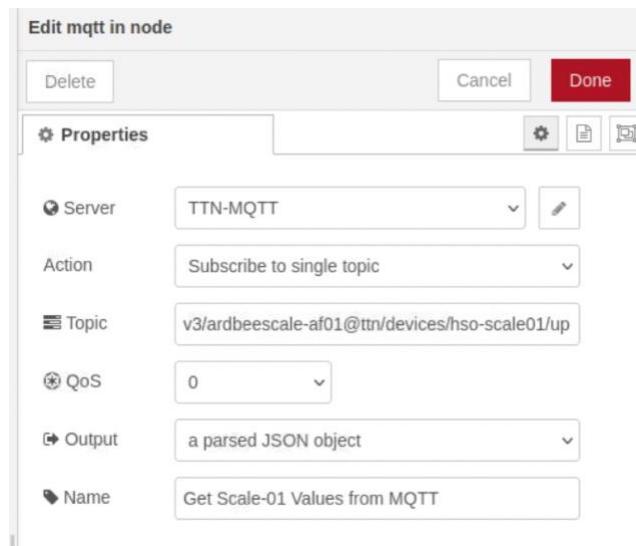
Video of explanation about influxDB [here](#). This is a « time series » data base. The table must be created manually and then the different tables (called measurements) are created as data are received.

5.2.1 Fetching uplinks in node-red

Doc on TTN [here](#) and following pages. Summary :

It is the TTN server that receives and decodes the payloads received from the hives. To retrieve them, TTN offers an MQTT interface and you can therefore subscribe to the topics corresponding to the hives to retrieve the payloads decoded into JSON files.

We can do this with an MQTT node from Node-red (node = function packaged in a piece of code and available in the form of a Node-red module) which allows to define the topics that you want to recover . When you double click on the node, you come to the "properties" page which allows you to configure it.



1st line : Server : click on the pen beside the field « server ». Several tabs are available :

Tab « Connection » :

- host : [eu1.cloud.thethings.network](#)
- Port : 1883
- Do not check « use TLS »
- Protocol : [MQTT v3.1.1](#)
- Client Id : [Leave blank for auto generated](#)
- Keep alive : [60](#)
- select « Use clean session »

Tab « Security » :

- User name : [ardbeescale-af01@ttn](#)
(syntax : application_name@ttn)
- Password : the one provided by TTN in the page « integration MQTT »

Tab « Messages » : default selections are OK

Second line : Topic. General form is `v3/{application id}@{tenant id}/devices/{device id}/up` which, in my case is : `v3/ardbeescale-af01@ttn/devices/hso-scale01/up`

Third line : QoS. TTN doesn't manage QoS. Put 0

4^{ème} ligne : Output. « auto detect »

5^{ème} ligne : the name given to the node.

If you want to use Mosquitto as a bridge (not very interesting nor useful) you can find explanations in the video [Bridge The Things Network to your local MQTT broker](#)

At the output of the MQTT node of node-red, we get the JSON file sent by TTN. We need to extract from it the values transmitted by the sensors. To do this, we use the "json" node of the "parser" category. Then, we connect the output of this node to as many "change" nodes as there are data types to store in the database (weight, temperature, etc...). The input to these "change" nodes is the JSON file from the previous node. Then we click on this "change" node and in the "rules" zone, we choose the "Set", "msg".payload options, and to fill in the "to" field we select "msg" and in the field next to it, we will paste the copied value into the debug window by clicking on the "copy path" icon when the field we want is selected.

5.2.2 Data storage in influxDB

5.2.2.1 Set-up and useful commands

First of all, we have to create the influxDB database (hsohives) which "measurements" (which are the equivalent of "tables" in relational database), are created automatically as data comes in. They will contain the different parameters sent by each hive. In portainer, we click on the icon > and we are in the internal terminal of the container. In my case, Portainer credentials are admin/Trcvx123%###

The commands to execute are :

```
influx to enter into influxDB editor
show databases to list of the databases already created
create db hshives to create the database
use hshives to select the database
show measurements : to list all the tables of the database
select * from myMeasurements limit 5 to list the 5 first elements of the table
drop database nom_base to delete the database
```

See the list of commands [here](#).

It is often useful to remove data. Typically when the scale has made a measurement while the hive was open, it introduces a low point that crushes the scale and impairs visibility. The first step is to identify the problem line:

```
select * from scale01weight where value<40000
```

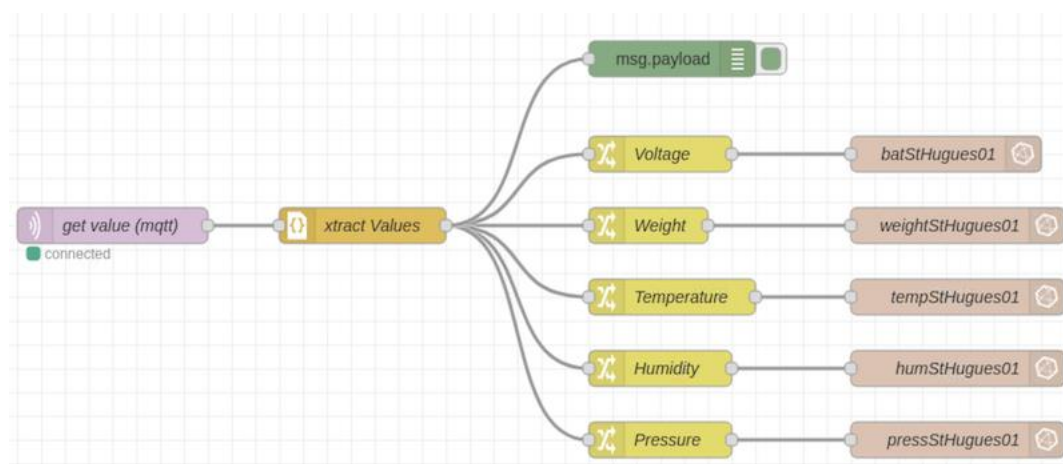
We then identify the time of the problematic line and delete it:

```
DELETE from scale01weight where time=1662653742651485372
```

5.2.2.2 Loading the data in influxDB with nodered

First we need to load the node "influxDB" in node-red. By double clicking on this node, it's possible to configure it. Three fields must be defined:

- The name of the node ("load Weight to influx")
- The server and its credentials (this is where the database name is specified and how to access it). This is done by double clicking on the little pencil in this field:
 - Name (this is an arbitrary chain of characters) : HSO scales
 - Version : 1.x
 - Host: instead of entering the IP of the host in the field "server", simply enter the name « influxdb » and 8086 as port.
 - Datadase : hshives
 - User name: admin
 - Password:
 - Don't select « enable secure connection »
- The last field to be entered is the name of the measurement in which the date will be stored ("scale01weight")



5.2.3 Sending downlinks

The method is explained in TTN doc [here](#) and following pages.

To send downlinks, we will use the dashboard, which is a set of nodes that allow us to view the data in the node-red user interface (accessible on <http://localhost:1880/ui>) with gauges, graphs, etc... and on the other hand to enter data to be sent to nodes (with buttons, slider, etc...). The dashboard also allows to define notifications rules.

If it is not already present, the dashboard can be loaded in Node-red (manage palette, node-red-dashboard).

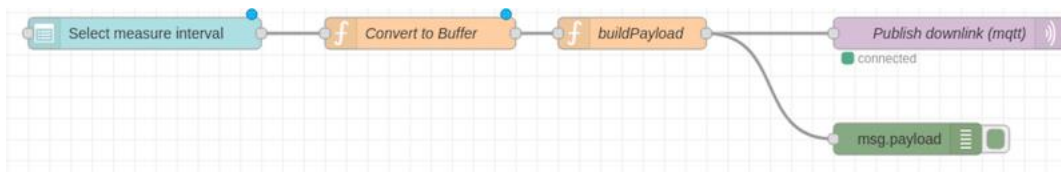
To send a payload to a node, we load a "button" which is configured with the payload we want to send. Be careful, the button must be part of a group which must itself be part of a tab; you have to create the group and the tab by clicking on the small pencil.

Then we connect this button to an "MQTT out" node that we configure so that it is connected and can publish on the mqtt broker of TTN.

Issue : the explanation of the TTN doc uses an "inject" node by configuring the data entered as a "buffer" (a type of variable from node.js, see explanations [here](#)) while the nodes available in the dashboard do not offer this option (the data entered can only be numeric). It is therefore necessary to introduce a "function" node which will transform the data entered into a "buffer". The code is the following:

```
var myArray = [msg.payload]; // transformation de la donnée en Array
var myBuffer = Buffer.from(myArray); // creation duBuffer
return {payload: myBuffer}; // on retourne le buffer
```

Then another "function" node must be added with the code proposed in the TTN doc and everything must be connected to a correctly configured mqtt_in node. We can then send downlinks from the node-red user interface.



5.2.4 Node-red dashboard on a smartphone

Search « remote-red » on app store. The developer proposes two explanatory videos: [vidéo 1](#) and [vidéo 2](#).

Summary : first you have to load (with manage palette) "remote-red-contrib-remote". Then we place in the node-red flow the node "remote access". We open this node and of course we have to configure it (protocol: http, Serving Port: 1880, Base URL: /UI) and we click in the app on "Connect another remote-red app", which displays a QR code that we have to scan by clicking, at the bottom of the screen, "Add node-red instance".

The app and node-red are then connected and you can act either from the app or from the node-red dashboard and you can see the same on node-red/UI and on the app.

The app cost is 6.49€/year.

After testing it, I am actually not using it and uses Blynk to send the most frequently used downlinks and a python program for the others.

5.3 Visualization with Grafana

Accessible by a browser on port 3000. Credentials : initially it is admin/admin, I changed it to admin/trcvx123

In the first step, we need to define the "data source" (icon "settings") where we must select InfluxDB and fill in the name of the database and its credentials. The other default fields are OK, except the one specifying the URL: instead of <http://localhost:8086> (which doesn't work; why ??) I had to put IP_ADDRESS:8086 where IP_ADDRESS is the ip address of the raspberry

About Grafana :

See the end of [this vidéo](#) where the author explains the key notions of Grafana.

Here is a very basic example of a Grafana chart:



Several charts can be combined on a so called "dashboard" for representing the hive data in the chosen from on the chosen duration.

6 HARDWARE & WIRING

6.1 Load cell & support frame

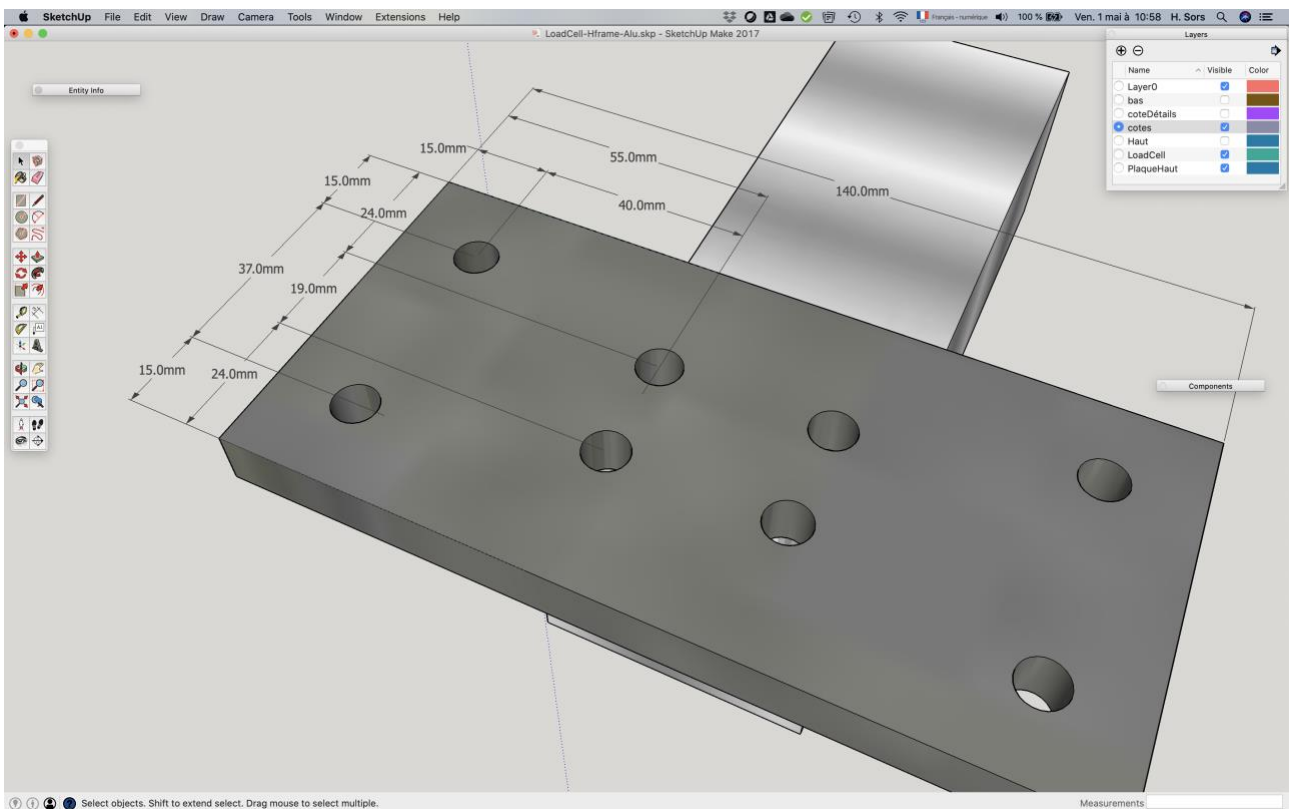
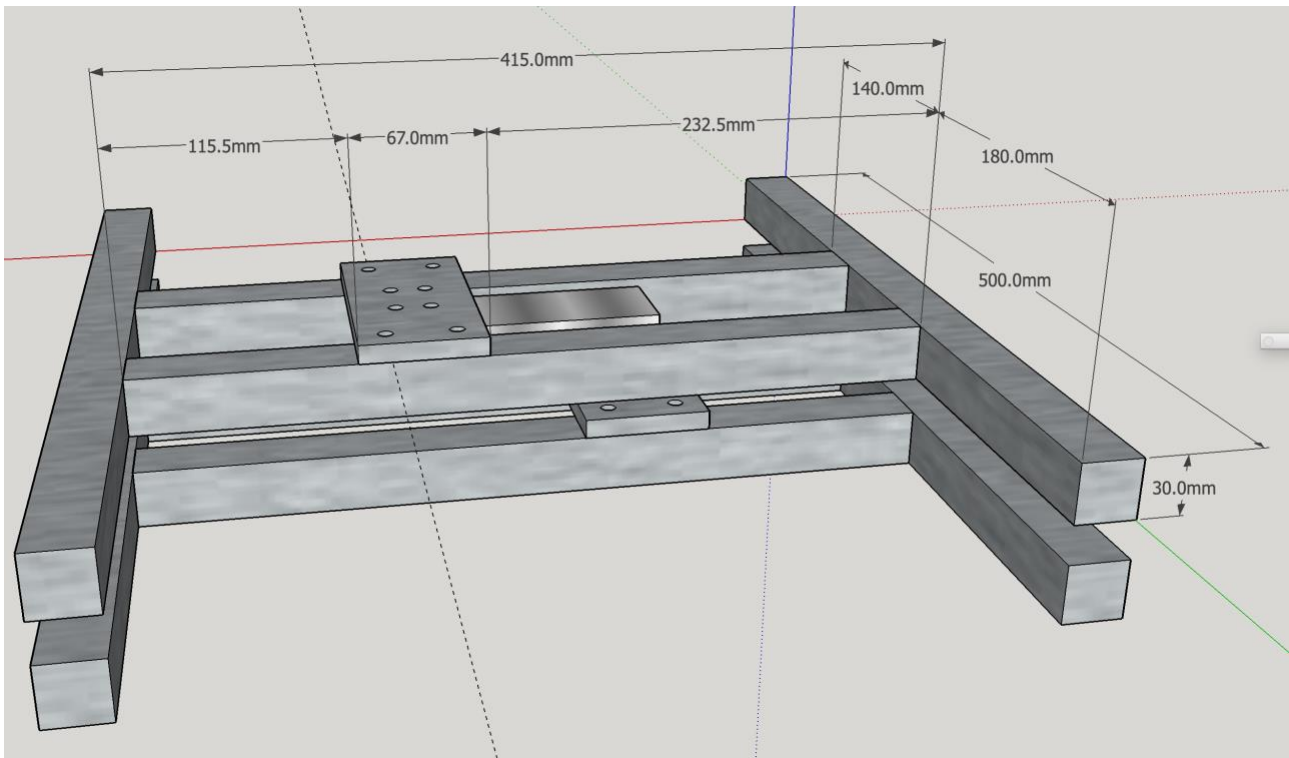
After testing several low cost load cells, I ended up using [a German made load cell « H40 »](#) more expensive (50€) but much better quality. The load cell is immobilized in a « Z » type of shape with two frame forming a « double H ». See picture here below. I have selected the 100Kg version but 150 or 200Kg may be more appropriate for very productive colonies.

I have a quote for a steel double H frame, which is ~75€ (excl. VAT) but I have finally opted for aluminum profiles that I bought on www.motedis.com. I created a kit containing all the necessary elements to assemble a load cell (except the covers to be mounted at the end of the profiles because they were out of stock). The kit is [here](#); its price before Covid was 56€ plus shipping (14€)

It's also possible to find what is needed on <https://www.technic-achat.com/> or another option is to buy the whole thing already assembled on 4bee.at where complete scale can also be bought. The frame only was at 120€...

With this type of load cell, stability over time is OK but weight drift with temperature can be a problem. I have therefore developed an algorithm to implement temperature compensation (see chapter [9](#)).

Below is the double H frame that I am using:



6.2 Wiring of the micro controllers and associated connectors

6.2.1 Wiring of BME280 sensor

You have to be careful when buying this sensor because it exists in two different versions with the same appearance: the BME280 which measures temperature, humidity and pressure and the BMP280

which does not measure humidity. AliExpress in particular maintains a clever confusion between the two.

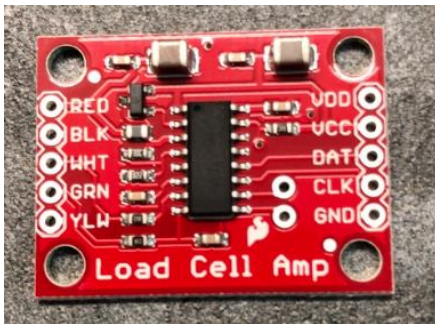
I bought the I2C version (4 wires to be connected) :

- Vcc,
- Gnd,
- SDA (data)
- SCL (clock)

The power consumption of the BME280 is supposed to be $3.6\mu\text{A}$ when reading every second and $0.1\mu\text{A}$ in sleep mode. So I decided to connect Vcc on the 3.3V of the $\mu\text{controller}$ and not on a GPIO pin which would have allowed to cut the power supply.

6.2.2 Wiring of the HX711 board

The IC is mounted on [module HX711](#) bought from Sparkfun. A JST5 connector is soldered on the side of the module to be connected to the microcontroller, while on the analog side the wires coming from the load cell are soldered directly on the board, in order to minimize possible connection problems.



The total power consumption of the HX711 is $1500\mu\text{A}$ in normal mode and $0.5\mu\text{A}$ in power down mode. It is therefore unnecessary to provide a GPIO pin to disconnect the power supply of this HX711, it is simply put in sleep mode the driver with the ad hoc command.

6.2.3 Wiring of the relay board

The relay module, needed to control the syrup pumps, has a quiescent current of several mA, which would drain the battery quickly. It is possible that this quiescent current could drop significantly if the LED is removed from the power supply but I did not try it. I preferred to supply the relay with the 12V (available from the PV panels) through a 7805 voltage regulator so as to be sure that the relay does not consume energy from the 3.3V battery.



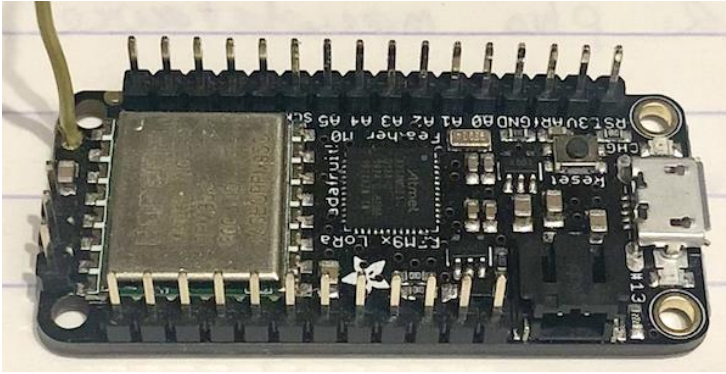
The relay is triggered by connecting GPIO10 directly to the relay control.

6.2.4 Microcontroller wiring

The microcontroller must be connected to the following elements:

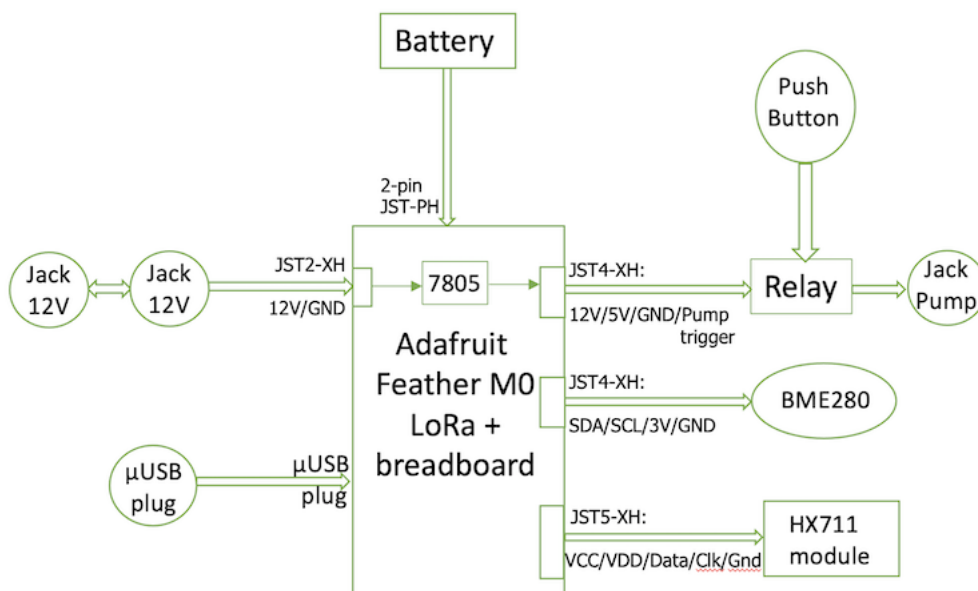
- The temperature/humidity/pressure sensor BME280, placed outside the box, under the hive, therefore in the shade, protected from the sun.
- The HX711 module connected to the scale
- The relay controlling the sirup
- It must also receive the 12V in order to provide 5V for the relay

The microcontroller is delivered with SIL male connectors for easy testing :



But this configuration does not allow to easily wire the elements that need to be connected to the microcontroller. So I used a soldering breadboard that I cut to the size of the microcontroller PCB and I solder female SIL connectors on it in order to plug this breadboard on the microcontroller. We can then use the surface of this small board to solder the connectors allowing to make the connections towards the outside as well as the 5V regulator feeding the relay and we can connect all that by soldering the wires on the breadboard. Knowing that the 12V is present on the breadboard, we bring it to the relay with the JST connector used to bring the 5V, the GND and the control signal of the relay to this relay.

Corresponding connection scheme is as follows :



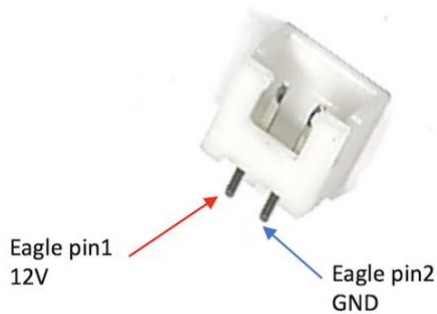
The elements represented by square boxes are inside the case and the elements represented in round boxes are elements accessible from outside the case. The box marked "push button" represents a push button mounted in parallel with the relay and allowing to manually activate the syrup pump in order to calibrate it.

To be noted: in the final version I have added another external connector which is not represented here above and which uses the internal spare connector that I have included on the final pcb design. It's a 1W temperature sensor allowing to measure brood temperature inside the hive. The pcb includes the solder pads of the connector but the connections have been made manually.

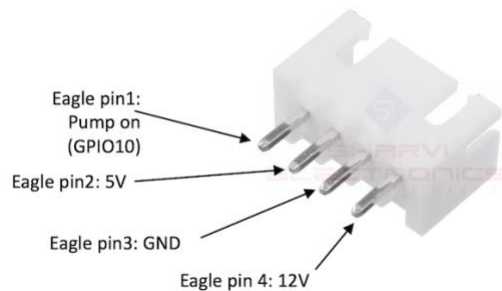
Warning : There is a small defect in the choice of connectors because the set includes two identical connectors (JST4) and it is therefore possible to make mistakes and make bad connections.

The pin-out of the different JST connectors is as follows (I have indicated the pin numbering of the JST connectors as the Eagle software considers them):

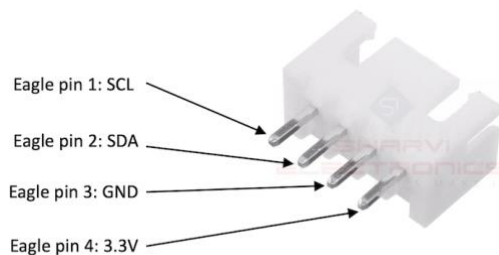
JST2 connector bringing the 12V :



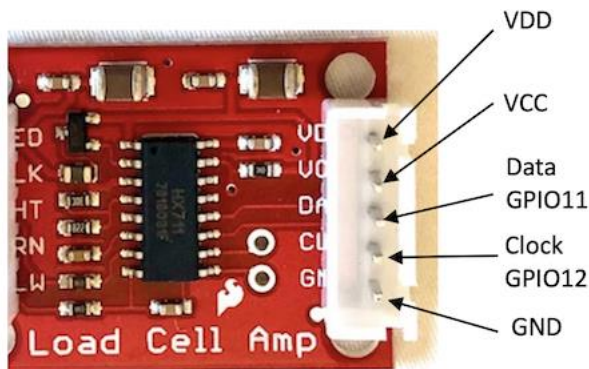
board to relay:



Board to BME2080 :



Board to HX711 module :

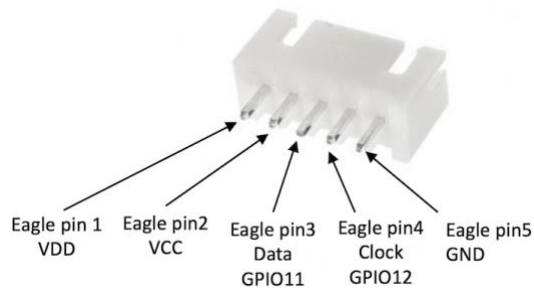


Vcc et Vdd are both connected to 3.3V of the Arduino (if 5V would have been available it would have been better to use it for Vcc to increase the measure precision).

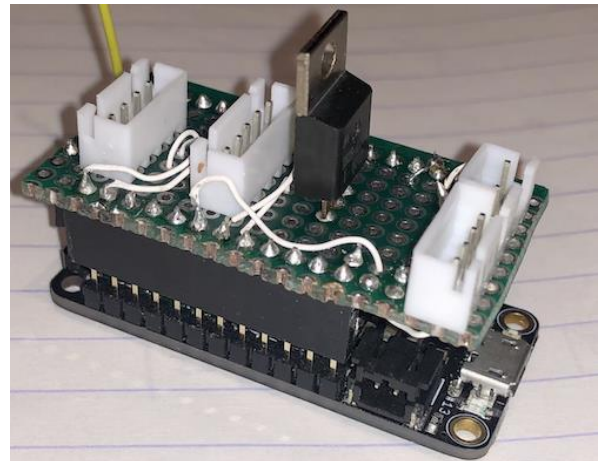
As the HX711 module is connected to the micro-controller by a JST5-XH male cable wired as shown opposite (which crosses the connections), the wiring of the JST5 connector on the micro-controller module must be reversed compared to that of the HX711 module

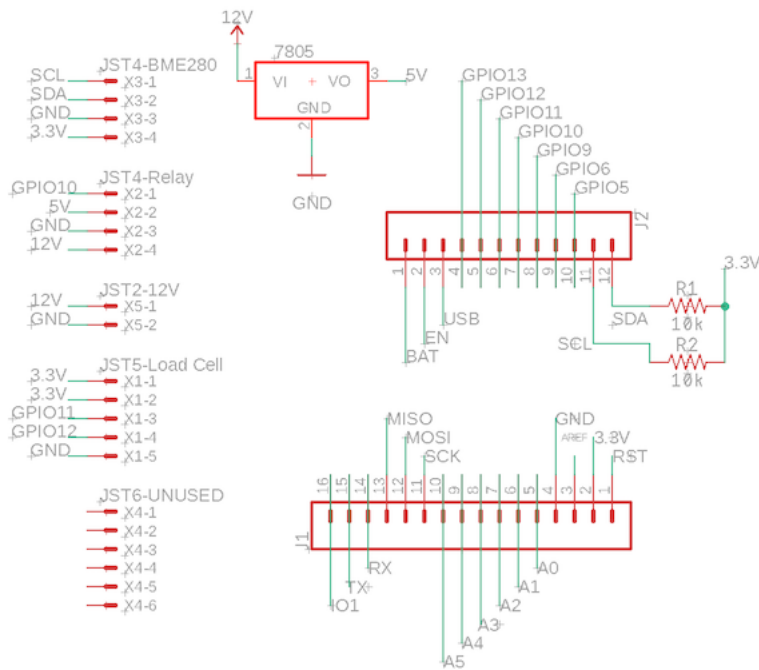
The HX711 module is connected to the micro-controller by a JST5-XH male-male which crosses the connections. The wiring of the JST5 connector on the micro-controller module must be reversed with respect to that of the HX711 module.

The pinout of the board connector will therefore be :

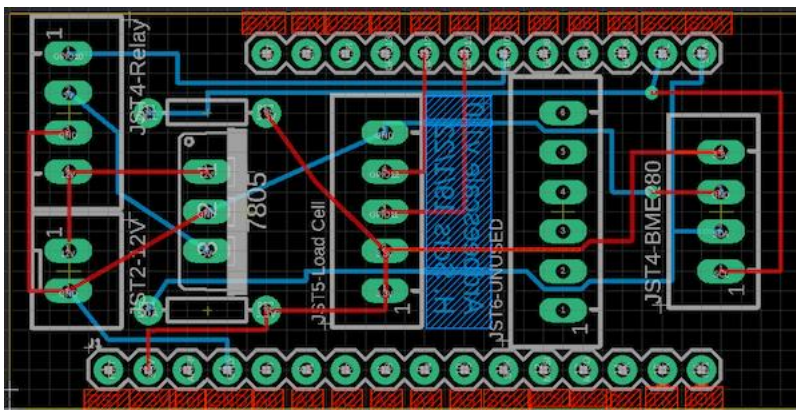


Initially I used a breadboard with all the wiring being done manually (not easy because there isn't much space) and at the end here is what I got. Most of the wires are underneath, not visible on the photo.

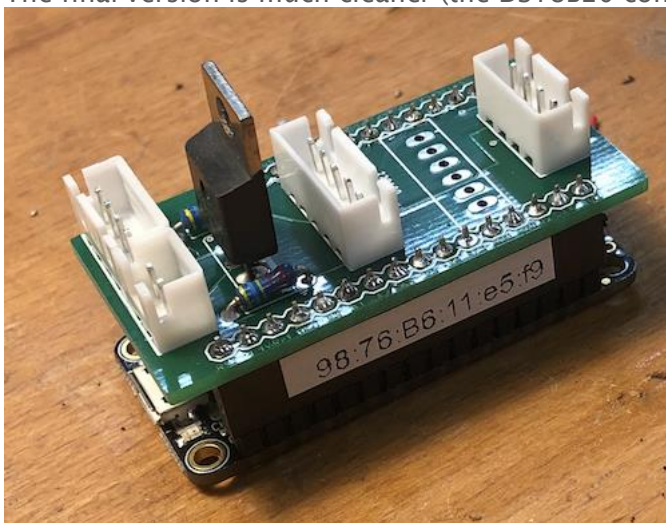


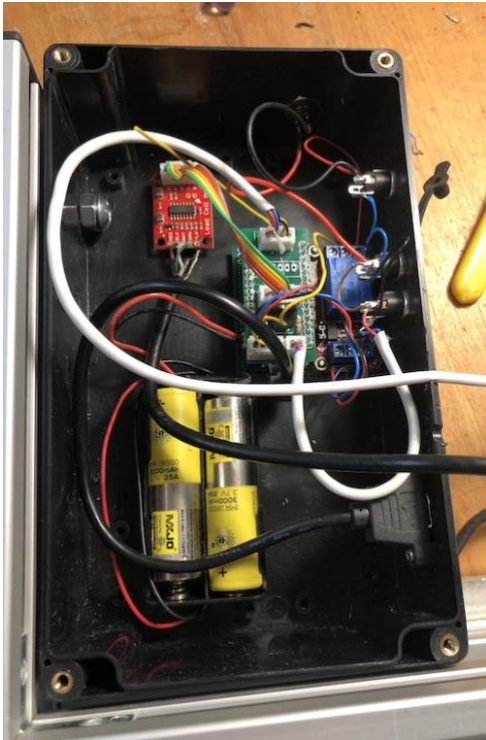


Finally, to avoid having to redo the wiring several times, I created a small board with Eagle software (free version) and I ordered the pcb from jlcpcb.com. The procedure is very simple: you upload the zip file generated by Eagle. The site visualizes the board and you just have to order it. Here are the Eagle electrical schematic and the PCB design (files plugFeather.brd for the board and plugFeather.sch for the electrical schematic).



The final version is much cleaner (the DS18B20 connector was not yet populated in this photo) :

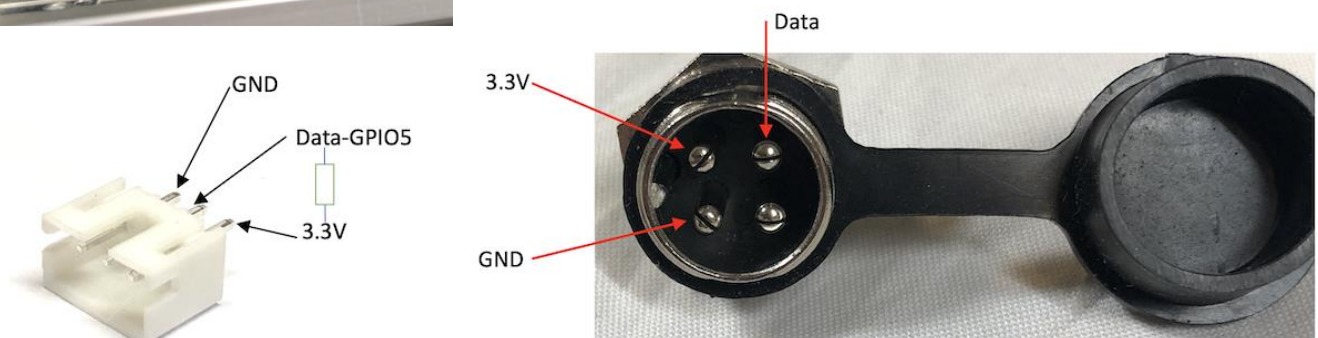




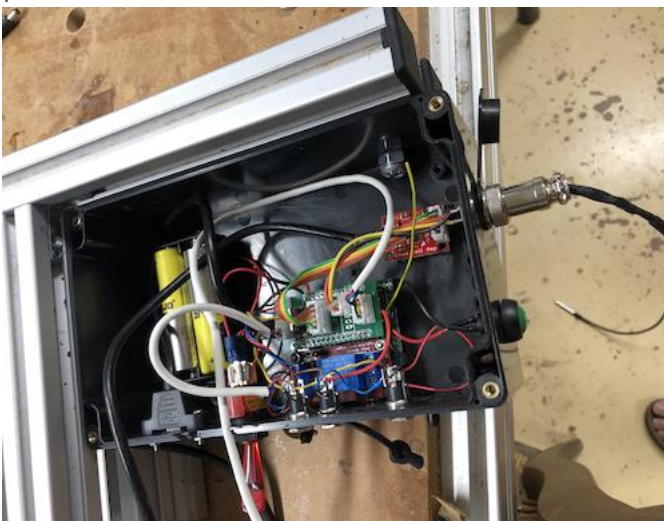
The microcontroller + connection board is fixed in the box with velcro, next to the other elements also fixed with velcro: the relay module, the HX711 boards and the battery.

Finally, I added a JST3 connector on the spare slot on the PCB in order to connect a DS18B20 temperature sensor to be placed inside the hive. This sensor uses the GPIO5 of the microcontroller and also brings the 3.3V and the GND on a connector fixed on the box next to the calibration button of the pumps. Again, this is not shown on this picture which was taken before the modification.

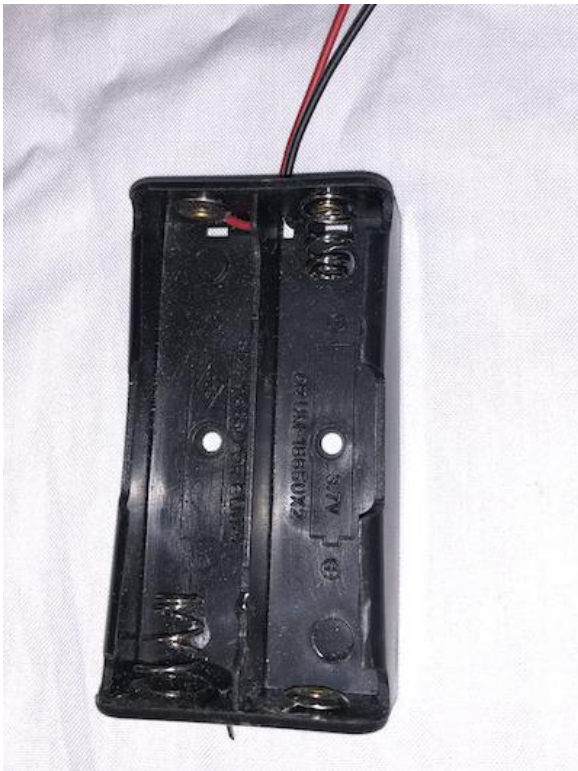
Corresponding connections are represented here below :



Here is the final version, with the DS18B20 sensor connected:



6.2.5 Modification of the batterie enclosure



I wanted to put two 18650 batteries of 3.7V/3000mA in parallel to have a sufficient autonomy but I didn't find a box allowing to put these two elements in parallel. So I bought the box on the left that puts them in series, then I cut the piece of wire that connects the + of one battery to the - of the other and I pulled a wire so that the two batteries are in parallel. The manipulation is a bit crappy because when I cut the wire that connects the + of one battery to the - of the other, the spring comes off and you have to glue it to hold it.

The result is far from perfect will do the job :



6.2.6 Charging the battery

The microcontroller embeds a charger that is powered by the 5V present on the USB connector (100mA charge, so 30H are needed to recharge 3AH...).

Knowing that the 5V is present on the add-on board plugged into the microcontroller (when an external 12V power source is connected to the device), it would have been simple to pull a wire that brings back this 5V on the USB connector so that the battery is recharged permanently when a 12V power source is connected to the device. But this could be a problem when a computer is connected to the USB port because it also provides 5V on the same pin.

So I decided to do otherwise: all my hives are connected to each other's in order to distribute the 12V power supply to all of them (reminder: this is needed for the sirup pimps). This is done thanks to the two jack female connectors mounted in parallel on each enclosure. The 12V is coming from the battery to the first hive with a cable plugged in one of those two jack connectors. And another cable is plugged to the second jack connector to carry the 12V supply to the next hive. A 7805 5V regulator is incorporated in the cable, drowned in a resin formed with hot glue, as shown on the photo here below and the output of this 5V regulator is connected to a micro-USB male connector that can be plugged in the micro-USB female connector of the enclosure.



Knowing that I have 12V power available in my apiary (coming from 2 PV panels charging a 12V battery), I can run my scales without battery, which may not be a bad idea, knowing that Li-Ion batteries do not like cold temperatures very much...

6.2.7 Content and choice of the enclosure

I have used [this box](#) bought on Amazon for 14€. It is a bit too big but that doesn't hurt and allow an easier access to the various elements it contains and also make the cabling easier.



Inside the box we have :



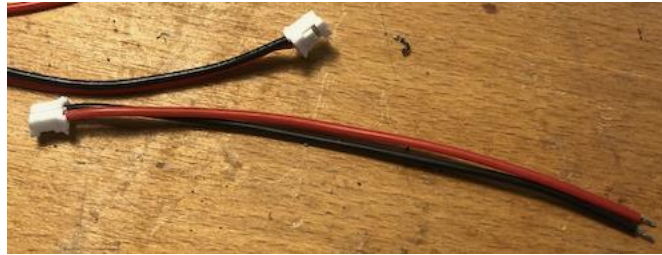
- The Adafruit microcontroller on which the following elements are connected :
 - The antenna (a simple wire)
 - A relay needed to trigger the sirup pump
 - A push button allowing to manually turn the relay on (very useful to prime the pump)
 - A cable bringing the ad hoc signals toward the BME280 sensor
 - A cable connected to the HX711 board
 - The battery
- The pump relay
- The HX711 board
- The battery
- A cable bringing the micro USB connector of the microcontroller on the enclosure front face so that it's possible to program (and recharge the batterie) without opening it.

Several connectors are present on the enclosure faces :

- A female jack connector (like [his one](#)) bringing the 12V from outside
- Another female jack connector similar (and connected in //) to the previous one allowing to output the 12V for the next hive beside.
- Another similar jack connector (separated from the two previous one so that no confusion between those is possible), brings the 12V to the sirup pump when relay is ON
- A micro USB connector purchased on [Aliexpress](#) connected to the micro USB connector of the Adafruit microcontroller (used to recharge the battery or reprogram the microcontroller).
- The load cell cable goes through the front face of the enclosure without any connector.
- Same for the BME280 sensor
- The push button allowing to calibrate the pump or to prime it when the sirup is introduced for the first time. Purchased on [Amazon](#).

For the antenna, I simply use a 8.62cm long wire (1/4 of the wavelength of a 868MHz signal - LoRa frequency in my area) soldered directly on the Adafruit board

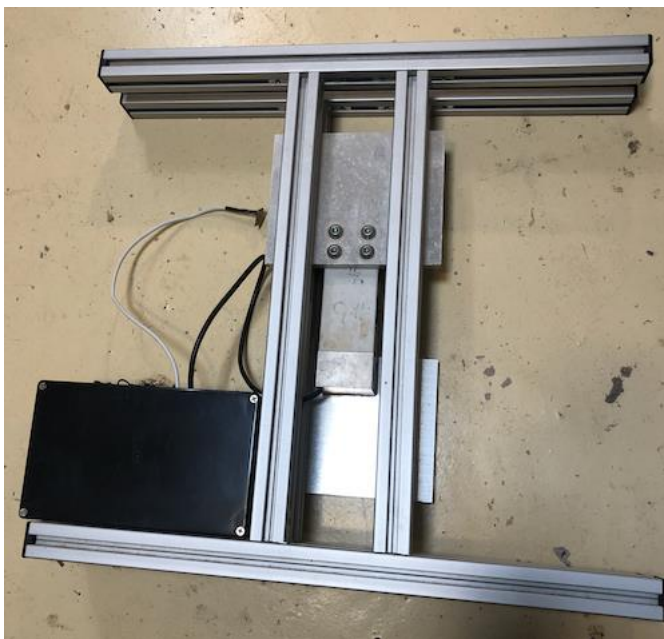
To power the unit, I use two [18650 modules](#) (3.7V, 3000mAH each) mounted in parallel in [this enclosure](#) purchased on Aliexpresset. As explained before I had to change the wiring so that the two batteries are in parallel and not in series. To connect the batteries to the microcontroller, I found on [Amazon](#) the ad hoc JST2 connectors connected to wires ready to be soldered:



To fix the various elements in the enclosure, I used scratch glued on the box.

6.2.8 Mounting the enclosure on the scale

The case bought on amazon is fixed on the aluminum frame thanks to a bolt screwed in a nut sliding in one of the branches of the aluminum frame (picture below). Be careful to put a washer between the plastic case and the aluminum frame in order to avoid any friction between the case and the frame (which could distort the measurement).



7 WEIGHT MEASURE ACCURACY

See [this note](#) about the ADC which explains all the possible errors impacting the measure.

The weight measurement is also affected by changes in the temperature of the sensor itself. The measured value varies linearly with temperature (with a fix weight on the scale) and it is therefore tempting to implement a simple linear correction of the weight measurement in order to get a temperature compensated weight. Unfortunately, the sensor mounted in the scale has a certain temperature inertia that is ignored by the linear compensation which, at the end, doesn't work very well. I have therefore developed a non-linear compensation model that works correctly; it is explained in details at the end of this document in chapter [9](#) (Project presentation & Technological choices [Weight temperature compensation](#))

8 HOW TO ACCESS A GOOGLE SHEET FILE FROM PYTHON SCRIPT

Compared to what I wrote in the `rasbeescale_v2.docx` file, the creation of the credential requires less details than before (no choice "web server" or "application data")

The Google documentation has been updated accordingly :

<https://docs.gspread.org/en/latest/oauth2.html>

The google cloud console to be used is <https://console.cloud.google.com/>

Below is the procedure as of 20/11/2022 (Google changes it regularly but the spirit remains the same)

- 1) At the top of the console, we see the existing projects and we can create a new one by clicking on the arrow:



You can access the dashboard by clicking on the menu on the left of Google Cloud Platform (the 3 horizontal bars) and choosing "API & services" then "Dashboard".

- 2) For this project, you need to Enable Google Drive API and Google Sheet API (by typing "Google Drive API" and "Google Sheet API" in the search field of the "API & Services" screen)
- 3) We find in API_services a Credential menu and we will create a credential of type "service account" with the role "owner". We give it the name of the apiary.
- 4) By clicking on the "service account" that you have just created, you access a menu (at the top) and in this menu you click on KEY, then on the ADD KEY icon. We choose the option "New Key" and as key type, json. The file is automatically downloaded to the computer
- 5) We can change the file name to put a simpler name: we decide to name these files `apiaryName_secret.json`. Take note of which email appears in this file
- 6) Put this file in the directory of the python script that will access the gsheets file (or reference this file with its full name in the script)
- 7) Share the gsheets file with the email that appears in the credentials file
- 8) The code needed to access the gsheets file is as follows:

```
import gspread
```

```
from oauth2client.service_account import ServiceAccountCredentials
```

```
scope = ['https://spreadsheets.google.com/feeds', 'https://www.googleapis.com/auth/drive']
```

```
credentials = ServiceAccountCredentials.from_json_keyfile_name('nomRucher_secret.json', scope)
```

```
gc = gspread.authorize(credentials)
```

```
wks = gc.open("A1-RasbeeData").sheet1
```

```
wks.update_acell('B2', "coucou ça marche")
```

The documentation on [github](https://github.com) and on the web site referred to gives examples on how to manipulate gsheets files. The list of API is available

Share the gsheets file with the email that appears in the credentials file on google site [here](#).

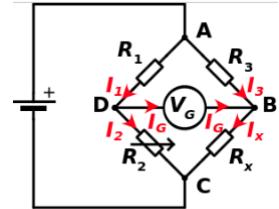
Warning, read <https://developers.google.com/sheets/api/limits>: there is a max number of free API call by unit of time. Beyond that limit, access is blocked until sufficient time has elapsed to respect the max number of call by unit of time.

9 WEIGHT TEMPERATURE COMPENSATION

9.1 Problem statement

This chapter contains all information necessary to understand why the weight measure is influenced by temperature and how to compensate the weight measurement to take into account the impact of temperature. Corresponding algorithm is implemented in the code for a 15mn time interval between consecutive measures. As of today in this version 1.0, the temperature compensation code doesn't adapt to a different measure intervals. It is optimized for a measure interval of 15mn.

The weight sensor is a strain gauge, which is a resistor which value varies very slightly when the metal in which it is embedded deforms under the effect of the weight to be measured. This resistor forms with 3 other fixed resistors of known values a "Wheatstone bridge". The circuit to which the strain gauge is connected measures the voltage difference V_g and then a simple calculation gives the resistance R_x of the strain gauge.



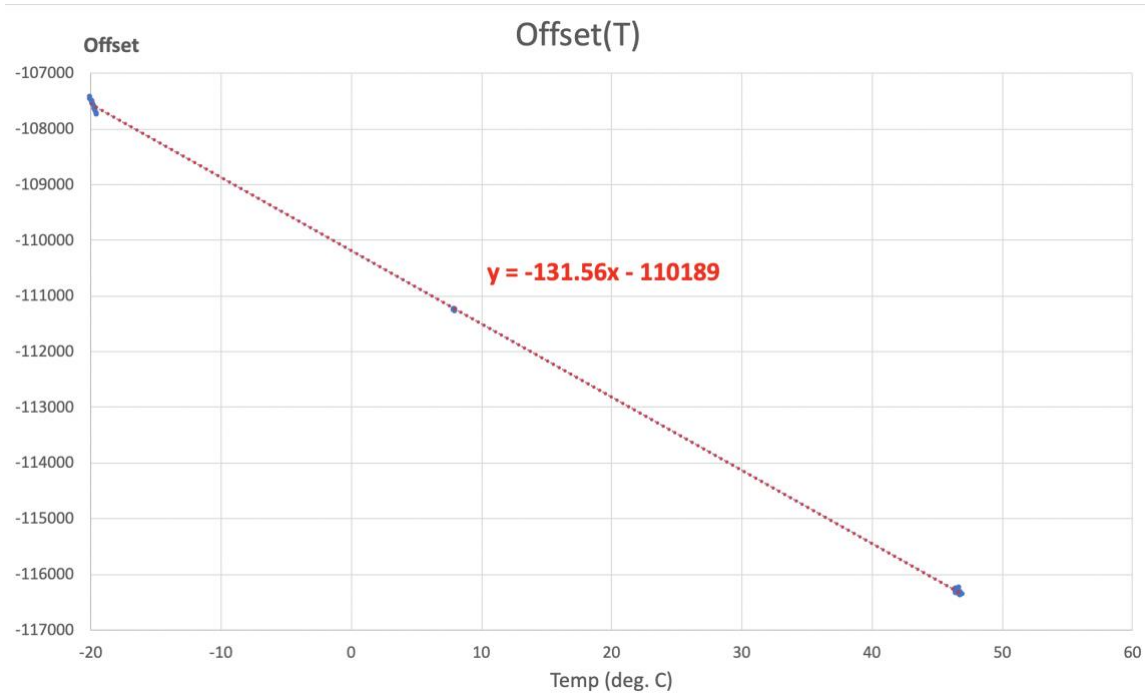
When no load is put on the scale, the readout of the ADC to which the Wheatstone bridge is connected is called the offset.

The output of the ADC when a load is applied to the scale represents the value of that mass, plus the offset.

Therefore, in order to measure a mass, we measure the output of the ADC when the mass is applied on the scale, from which we subtract the offset and the result is divided by a fixed factor representing how much the ADC readout value varies for 1g variation of the load applied to the scale. Load cells providers state that their products are "temperature compensated", which means that this difference (value read when load is applied minus value read when no load is applied) is more or less independent of temperature. But the value of the offset itself IS temperature dependent. According to my measurements, it varies linearly with the temperature with a slope which value depends on the load cell. I tested six different H40 load cells from the German supplier Bosche (by far the best supplier I have tested) and I was able to measure slopes that varied from 1 to 7g per degree. When the temperature fluctuates 20 degrees, the best load cell drifts by 20g, which seems acceptable and probably does not require implementing temperature compensation. On the other hand, the worst one drifts by 140g for this temperature range, justifying a temperature compensation.

We will see in this document how to implement an efficient temperature compensation based on a proposed modeling of the load cell and corresponding mathematical calculations.

At the end this document, we will apply the proposed temperature compensation formulas to actual data collected on a beehive scale and will show that temperature drifts can be almost entirely compensated thanks to the proposed model.



The graph above shows the variation of the offset measurement as a function of temperature for a “average” load cell among the six that I have tested. In this particular case, 45 units correspond to 1g so the offset varies 2.85g per degree.

On a “normal” scale, the variation of the offset with the temperature does not pose a problem because offset is measured each time power is switched on; the value of the mass minus the offset is therefore independent of the temperature. But “zero-ing” the offset at each measure is not possible with a hive scale because the mass is applied permanently and the value of the offset that is subtracted from the measurement is an offset measured beforehand at a temperature different from the actual measurement temperature. .

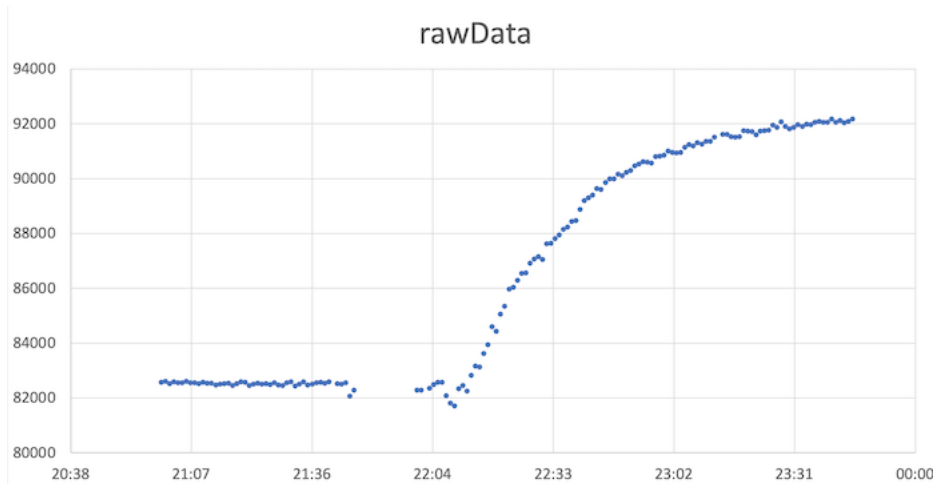
If the load cell is calibrated beforehand and we know how the offset varies with temperature (linear curve here above), we should be able to correct the measurement by calculating the true offset at the current temperature instead of the one measured beforehand at a different temperature. Unfortunately the load cell has a certain thermal inertia and when the outside temperature varies, the temperature of the load cell does not vary by the same value. We must therefore find a way to determine what is the internal temperature of the load cell based on historical temperatures.

9.2 Proposed nonlinear model

The load cell, especially when fixed in its scale structure, has a certain heat capacity C . Its temperature, which is assumed to be homogeneous inside the load cell, does not vary in the same way as the outside temperature because the load cell mounted in the scale has a given thermal conductivity σ which is not infinite. The quantity of heat dq that is absorbed by the load cell during a time dt causes its temperature to change by $\frac{C}{\sigma} \times \frac{dq}{dt}$, and it's easy to understand that the higher the temperature difference between the load cell T_{lc} and the outside temperature T_{ext} the faster the load cell temperature changes.

For an electronics engineer, all of this reminds what happens when a capacitor is charged through a resistor from a constant voltage source.

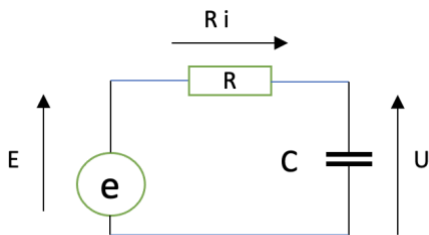
And as expected, when a temperature step is applied to the load cell mounted in the scale, we can observe that the ADC output evolves over time with a curve that looks a lot like the charge of a capacitor C (here the heat capacity of the scale) at constant voltage (here the temperature step ΔT) through a resistor (homogeneous with the inverse of the thermal conductivity: $1/\sigma$).



The graph above shows the change in ADC output when the unloaded scale is moved from an indoor environment with a stabilized temperature of 21°C to an outdoor environment with a temperature of -1°C (almost) stable over the duration of the measurement.

We now need to put all that in equations...

In the analogy with the evolution of the voltage U measured on a capacitor C when it is charged at constant voltage E through a resistor R , the diagram is as follows:



The charging current i is equal to the amount of energy stored by the capacitor per unit of time, i.e.

$$i = \frac{dq}{dt} \text{ and therefore, knowing that } U = qC \text{ the current is } i = C \frac{dU}{dt}$$

We can therefore express the voltage E as follows:

$$E = RC \frac{dU(t)}{dt} + U(t)$$

In our case, R is homogeneous with $1/\sigma$, the inverse of the conductivity, E is homogeneous with the variation of the offset resulting from the temperature step if the conductivity were infinite (let us call ΔO_0 this offset value), C is homogeneous with the heat capacity of the load cell and U is homogeneous with the actual measured offset $O(t)$. The differential equation which defines the temperature is therefore:

$$\Delta O_0 = \frac{C}{\sigma} \frac{dO(t)}{dt} + O(t)$$

$O(t)$ being the offset increment since the temperature step was applied

It can be demonstrated in mathematics that the solutions $y_k(x)$ of a differential equation of order 1 with constant second member, of type $y'(x) - ay(x) = b$ are of the form:

$$y_k(x) = ke^{ax} - \frac{b}{a}$$

k being a real number that is determined in physics with the initial conditions. In our case, the differential equation can be written:

$$\frac{dO(t)}{dt} + \frac{\sigma}{C}O(t) = \frac{\sigma}{C}\Delta O_0$$

So we have :

$$a = -\frac{\sigma}{C} \quad \text{and} \quad b = \frac{\sigma}{C}\Delta O_0$$

And the solutions to our differential equation are therefore of the form:

$$O_k(t) = ke^{\frac{-\sigma}{C}t} + \Delta O_0$$

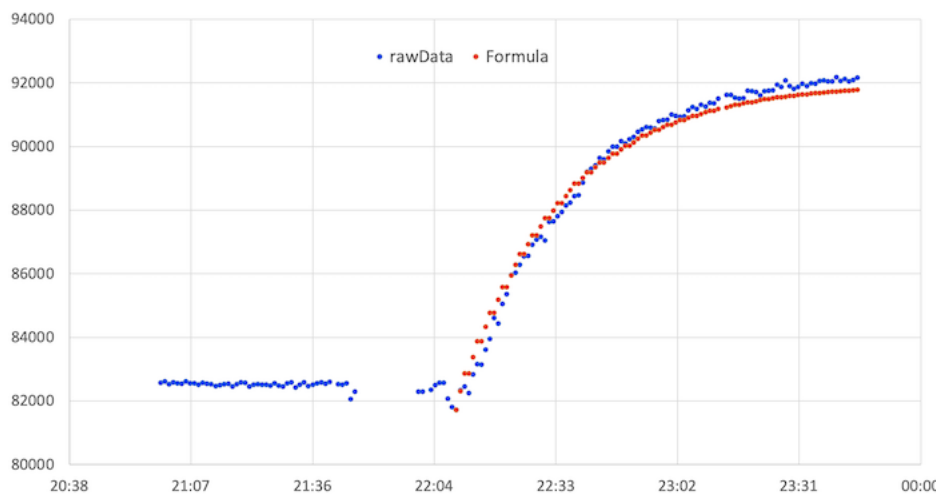
We know that at time $t = 0$, the increment of the offset is zero and we can therefore deduct that

$$k = -\Delta O_0$$

And therefore, the offset increment since the time the temperature step was applied is calculated as follows:

$$O(t) = \Delta O_0(1 - e^{\frac{-\sigma}{C}t})$$

The time constant $\frac{\sigma}{C}$ is unknown; we will therefore determine it graphically by representing on the previous graph the equation obtained above. We see on the graph below that with a time constant equal to 17 (the time being counted in minutes), the two curves overlap almost perfectly, which seems to indicate that the chosen model is correct.



By the way, offset being proportional to weight, the previous equation can also be written as:

$$W(t) = \Delta W_0(1 - e^{\frac{-\sigma}{C}t})$$

Which means – it's important to be precise because the notation is a bit misleading – that when a temperature step corresponding to a weight readout change of ΔW_0 (after an infinite time) is applied, the actual readout of the weight is evolving over time according to the above equation, knowing of course that the weight itself is stable over time.

But we are not done yet. The experiment and the calculations that we have just made have allowed us on the one hand to verify that the chosen model is correct and on the other hand to determine graphically what is the time constant of the scale measurement process, but we still do not know not how to calculate the temperature of the load cell based on the outside historical temperatures, which of course are varying over time instead of being constant as in our experience...

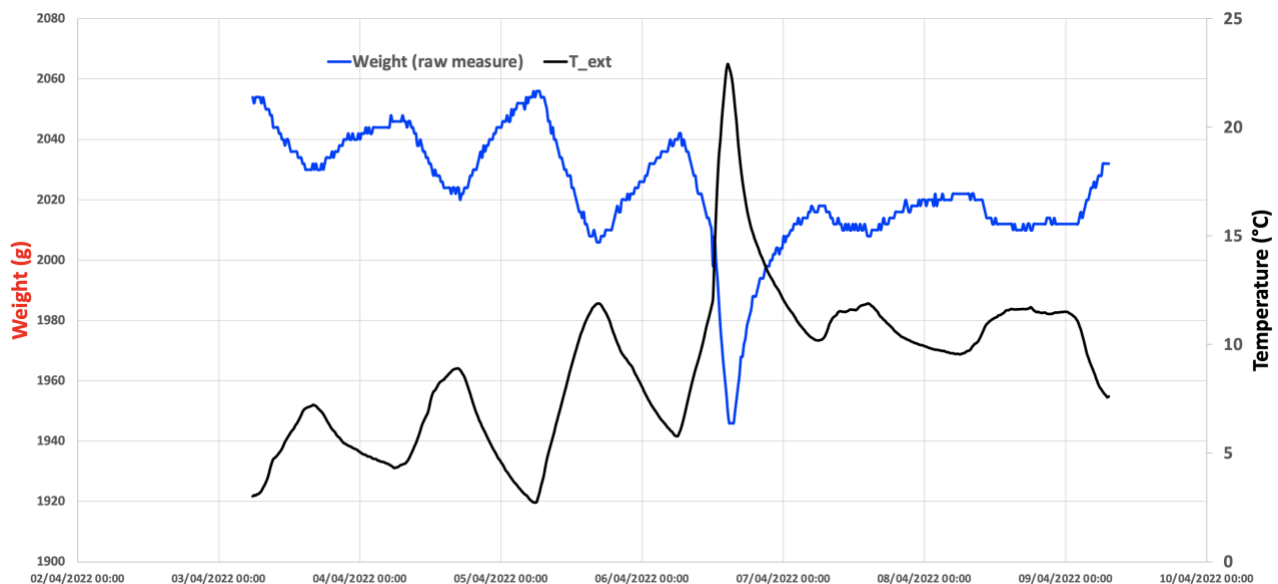
Here we need to make an approximation and we will assume that the temperature evolution over time is a series of small temperature steps applied each time we take a weight measure. In others terms, if we take a measure every 15mn, we will assume that the load cell temperature when we take a measure is resulting from :

- a temperature step applied 15 minutes ago and equal to the temperature difference between current measure and previous one
- another temperature step applied 30mn ago and equal to the temperature difference between the measure we took 15mn ago and the one we took 30mn ago
- plus the one before, etc, etc...

Because we know the time constant is 17mn, we will consider only the last 5 measures – which means that anything that happened before 75mn ago is ignored.

In order to test this model, we have put the scale in a closed box (to make sure temperature was correctly measured, avoiding the effect of sun shining on the scale) with a ~2Kg weight on it and we put this box outside during a week. Then we collected the raw weight measurement every 15mn along with the temperature at the same time.

Here is the corresponding graph:



The weight measurement when a constant mass is applied on the scale does varies quite a lot with temperature. We have seen before that weight readout is proportional to temperature and by looking at the extremes of the above graph, we can deduct what is the slope α of corresponding linear curve $W(T) = \alpha T$

In this case, $\alpha = 5.5 \text{ g/}^\circ\text{C}$

So when a temperature step ΔT_0 is applied on scale, the weight measurement changes – after an infinite time where temperature stays constant – by a value $\Delta W_0 = \alpha \Delta T_0$

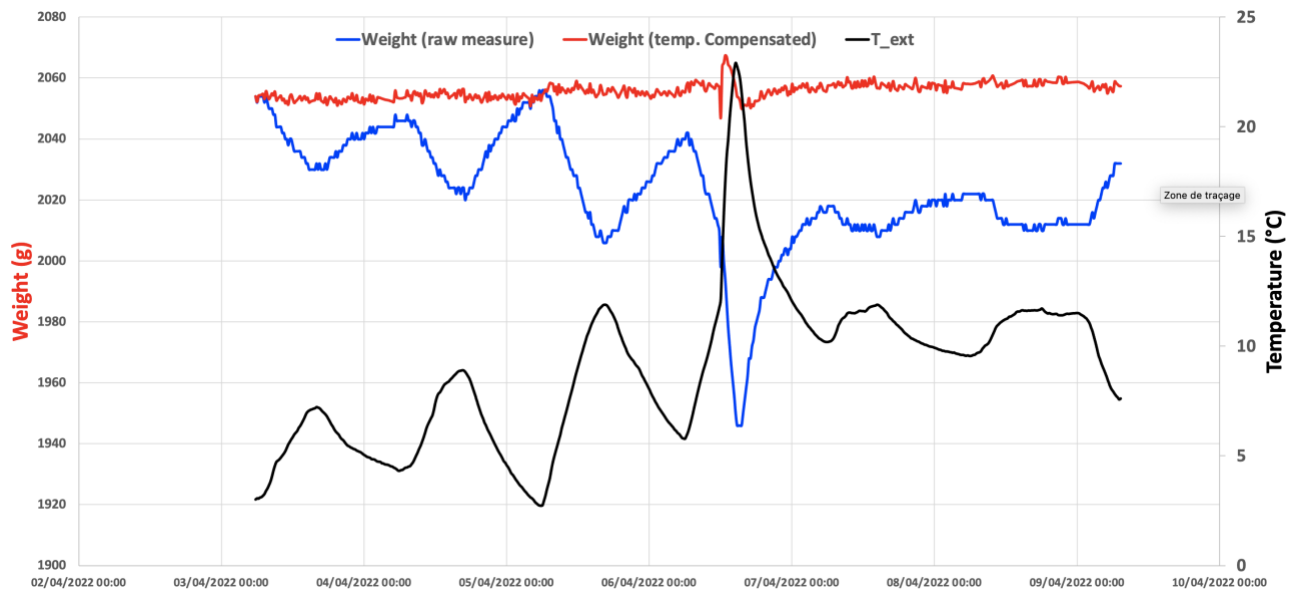
The previous equation can therefore be written $W(t) = \alpha \Delta T_0 (1 - e^{-\frac{g}{c}t})$

So each time we take a new measure, we applied this formula with $t = 15\text{mn}$, ΔT_0 being the temperature difference between this measure and the previous one, 15mn ago. This is giving us the value by which the weight measurement has changed as a result of this temperature step. To this value we add the weight measurement change resulting of the temperature step applied 30mn ago, plus the one 45mn ago, plus the one applied 60mn ago plus the one applied 75mn ago and we stop here because further in the past, the effect of a temperature step vanishes almost entirely.

The process is therefore the following: each time a weight measurement is taken, the above process is applied and the calculated weight measurement change is subtracted from the measured weight, giving a temperature compensated weight measurement.

The graph here below is the same as the previous one, on which the red curve representing the temperature compensated weight is shown. This weight is staying relatively flat when temperature changes, which seems to indicate that the model and the way it is applied is working OK.

The glitch in the middle corresponds to a move of the box in which the scale was placed from shadow to sun (as indicated by the sudden temperature increase). It's likely that this move has induced some false measurement.



Apart from the glitch resulting from a move of the scale, we now have a weight measurement system that is relatively stable when temperature changes.

Practically, the key to have a good weight measurement is to make sure the temperature is correctly measured, which is not a simple task (as soon as sun shines directly on either the scale or the temperature sensor, it creates bad measures).

9.3 Implementation

Here is how the code implements this temperature compensation.

Based on what we have explained before, we need to keep memory of all past external temperature data that can have an impact on the load cell temperature, given its thermal inertia. Knowing we take a readout of temperature at each measurement, we need to decide on how many past measurements we keep memory of the temperature. Arbitrary and knowing on one hand that our load cell mounted to its frame has a time constant of 17mn and on the other hand that most of the time we will choose to take a measure every 15mn, we have decided to keep the last 5 measures. With our parameters, the temperature of 5 measures ago will have 99% finished to influence the current temperature of the load cell. Longer interval between measures won't make any problem but shorter might.

Three arrays are declared in the code for the calculations:

```
double deltaText[5] = {0,0,0,0,0}; // an array containing the last 5 external temperature changes
double deltaTlc[5] = {0,0,0,0,0}; // an array containing the last 5 load cell temperature changes
double Text[6] = {T_EXT_INIT,T_EXT_INIT,T_EXT_INIT,T_EXT_INIT,T_EXT_INIT,T_EXT_INIT}; // an array containing the last 6
measurements of external temperatures
```


Each time a new temperature measure is taken, two arrays (`Text` and `deltaText`) are right shifted (the last value is dropped, and the first one is updated according to this last measure):

- `Text[0]` is loaded with the last external temperature measure,
- `deltaText[0]` is loaded with the difference (current temperature minus temperature at previous measure)

`deltaT_LC` is then calculated based on external temperature changes and time elapsed since those temperature changes:

- `deltaT_LC[0]` is loaded with the load cell temperature change resulting from a temperature step equal current temperature minus previous measurement temperature:

$$\Delta T_{LC} = \Delta T_{ext}(1 - e^{-\frac{\sigma}{c}t})$$
 where t is the time difference between current measure and previous one and $\frac{\sigma}{c}$ is the inverse of the time constant (`TIMECONSTANT`) as specified in file `scale-parameters.h`
- `deltaT_LC[1]` is loaded with the same calculation replacing t by $2t$ and taking into account the previous temperature change
- And so on and so forth until is `deltaT_LC[4]` calculated

Based on the calculations that we have explained earlier, the actual load cell temperature is the external temperature $5*t$ minutes ago (t being the time difference between two consecutive measures) plus the sum of each element of this array `deltaT_LC`.

`hive-parameters.h` file contains a few additional important parameters: `WEIGHTSLOPE` is one and is the value (in grams) by which the weight measurement varies when the load cell temperature increases by 1 degree (should be measured as part of the calibration process by taking note of weight measurement difference - with the same mass applied to the scale - at two different temperatures, differing from each other as much as possible, for precision sake).

As a result of the calculations we have just made, we now have the actual load cell temperature; if we know the initial load cell temperature at which the scale calibration was done, we can calculate the difference between those two temperatures, multiply this difference by the parameter `WEIGHTSLOPE` and this value (`current_temp - calib_temp`)*`WEIGHTSLOPE` would be the amount that we need to subtract from the last weight measurement to get the actual (temperature compensated) weight.

The load cell temperature at which the calibration was done is also a constant that is stored in file `hive-parameters.h`. Its name is `CALIBTEMP`. In real life, if this temperature is not known precisely, it can be approximated because in fact we don't care too much about the precision of this value because we are more interested in weight variation than in absolute weight (in other terms it's not a big deal if we make an error of a few grams, as long as this error doesn't vary over time and doesn't hinder the compensation calculation).

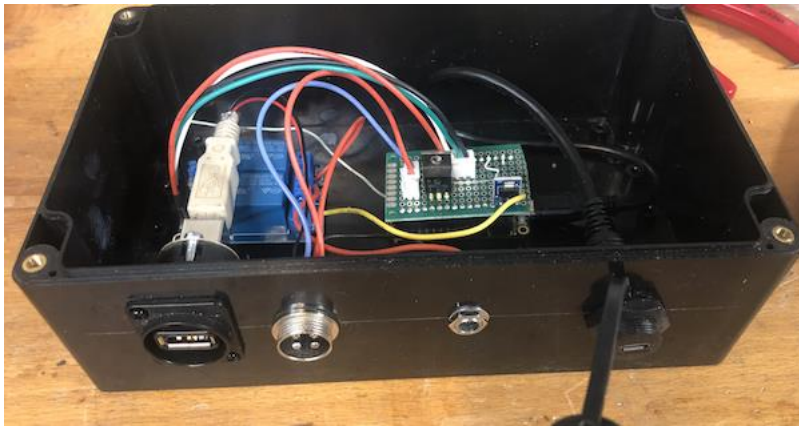
10 POWER SAVING UNIT

10.1 Introduction

In my implementation, the LoRa WAN gateway as well as the 4G key, are powered by a 12V battery which is recharged thanks to PV panels. In summer when sun is shining sufficiently and periods of dark sky are short, this is OK. However power is missing after a while in winter in case of bad weather and this is the reason why I have developed a simple power saving device which allows to switch on the Gateway and/or the 4G key only when the scales are sending data to the network.

In order to do that, it is necessary that all the devices are synchronized and share the same time. This is the reason why a "set time" command has been implemented in the code as explained in section [2.1.9 \(Main Sketch;\)](#)

10.2 About the hardware



The hardware design of this power saving unit uses the same micro controller Adafruit Feather M0 LoRa in the same enclosure as those of the scales. The micro controller receives an add-on board that I have wired manually, as pictured here.

The connector on the right is a micro USB one which is connected to the micro USB connector of the controller, allowing an easy reprogramming w/o opening the box

Next one is a 12V coaxial connector bringing the power (12V) to the gateway when needed

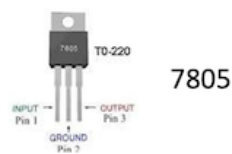
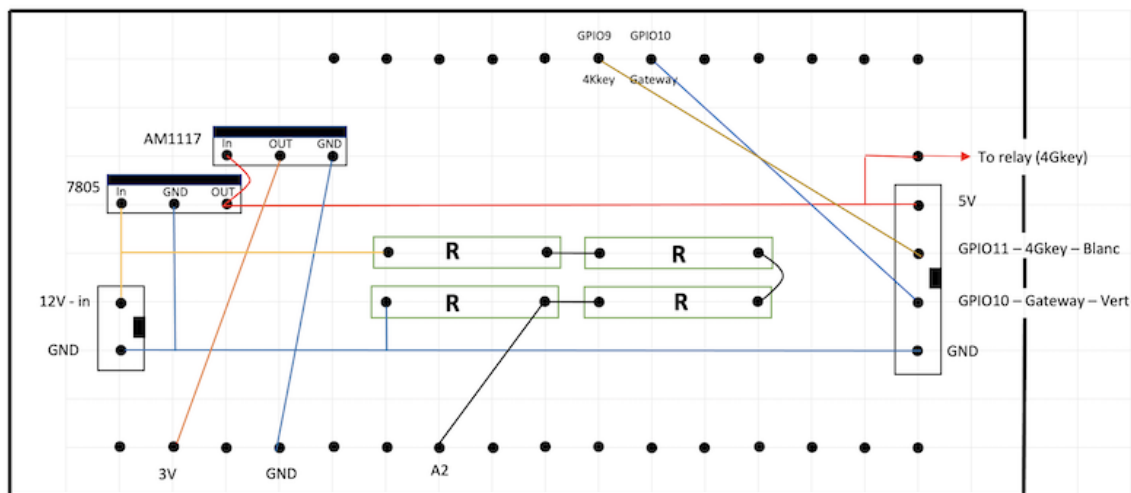
Next one is bringing in the power (12V) from the photo voltaic panels

The one on the very left is a female USB-B connector in which the 4G key will be plugged. Powered with 5V through a relay that is switched on when needed. In my case the outdoor gateway that I am using (Dragino DLOS8) has never been able to connect to the access point formed by the 4G key; I had to use the 4G option of Dragino, insert the SIM card inside it and configure accordingly, which means that this USB port is useless and therefore I have changed the code so that it is never powered up, which has the side benefit of saving some power (70mA consumed by the relay).

The add-on board plugged into the micro controller includes:

- A 7805 power regulator making the 5V that is used for powering the USB female plug (4G key) through a relay (allowing to switch on and off the 4G). It is also used by the 3V voltage regulator.
- An AMC117 power regulator making the 3.3V for the micro controller from the 5V of the 7805
- A 2 pins JST connector bringing in 12V
- A 4 pins JST connector connected to two relays: one for switching on/off the gateway and the other for switching on/off the 4G plug. GPIO10 is used for the gateway relay and GPIO11 for the 4G key relay.
- A resistor divider allowing to connect a fraction of the 12V input to A2 analog input. The sketch will read this input and send it as an uplink to the server so that it can be displayed on a dashboard.

Here is the wiring of this add-on board (top view):



10.3 Software

The code is mostly the same as the one of the scales with following differences:

- Everything that is linked to weight measurement as well as temperature/humidity/pressure is removed from the code
- Each time it wakes-up, it sends a two bytes uplink containing the battery voltage and the last downlink that was received
- The header file "hive-parameters.h" is replaced by "parameters.h". Everything that is linked to the scale itself is removed, plus:
 - PUMPTRIGGERCMD command is replaced by TIMEON. Same value: 00. This is the command portion (2 msb) of a one byte downlink.
 - If the most significant bit of the remaining portion is 0, the rest of this remaining portion gives a number (max 31 because it's on 5 bits) called earlyON meaning that the device must wake-up earlyON minutes because the normal "round time" at which the scales wake-up.
 - If, instead of 0, the most significant bit of this remaining portion is 1, then the rest of this remaining portion is called prolongedON, which is the number of minutes this devices will remain ON after the next coming "round time".

As an example, if measureInterval is 15mn and earlyON=3 and prolongedON=2, the device will wake-up 3mn before each quarter of an hour (at exH12MN ,xH27MN , xH42MN and xH57MN) and will stay ON 2 minutes after each quarter of an hour (=until xH02MN ,xH17MN , xH32MN and xH47MN).

Both relays are switched ON when the device wakes-up and OFF before it goes to sleep.

- STOPMEASURESCMD command is replaced by ALWAYS ON. Same value 11 (binary). This is the command portion (2 msb) of a one byte downlink.
 - If the value portion of this downlink is 0, it means the gateway must always remained ON (will not be switched OFF before the device goes to sleep).

- If the value portion of this downlink is 1, it means the gateway will be switched OFF before the device goes to sleep
 - If the value portion of this downlink is 2, it means the key 4G must always remained ON (will not be switched OFF before the device goes to sleep).
 - If the value portion of this downlink is 3, it means the 4G key will be switched OFF before the device goes to sleep
- EARLYON defines the default value of earlyON variable. Can be overridden by send sending the appropriate downlink (see below)
 - PROLONGEDON defines the default value of prolongedON variable. Can be overridden by send sending the appropriate downlink (see below)

As an illustration, following are the list of possible downlinks:

b7	b6	b5	b4	b3	b2	b1	b0	Décimal	Hexa	Fonction
0	0	0	x	x	x	x	x	**	**	TIMEON command. earlyON xxxx mn
0	0	0	0	0	0	1	1	03	03	TIMEON command: earlyON 3mn
0	0	1	x	x	x	x	x	**	**	TIMEON command: prolongedON xxxxx mn
0	0	1	0	0	0	1	0	34	22	TIMEON command: prolongedON 2mn
0	1	0	0	0	0	0	0	64	40	measureInterval = 1mn
0	1	0	0	0	0	0	1	65	41	measureInterval = 2mn
0	1	0	0	0	0	1	0	66	42	measureInterval = 5mn
0	1	0	0	0	0	1	1	67	43	measureInterval = 10mn
0	1	0	0	0	1	0	0	68	44	measureInterval = 15mn
0	1	0	0	0	1	0	1	69	45	measureInterval = 30mn
0	1	0	0	0	1	1	0	70	46	measureInterval = 1H
0	1	0	0	0	1	1	1	71	47	measureInterval = 2H
0	1	0	0	1	0	0	0	72	48	measureInterval = 6H
0	1	0	0	1	0	0	1	73	49	measureInterval = 12H
0	1	0	0	1	0	1	0	74	4A	measureInterval = 24H
1	0	0	0	0	0	0	0	128	80	Reset last downlink
1	1	0	0	0	0	0	0	192	C0	Never switch OFF gateway
1	1	0	0	0	0	0	1	193	C1	Gateway is OFF when device is sleeping
1	1	0	0	0	0	1	0	194	C2	Never switch OFF 4G key
1	1	0	0	0	0	1	1	195	C3	4G key is OFF when device is sleeping

The code will wake-up at “round time” minus earlyON minutes. It will then turn ON the two relays powering respectively the gateway and the 4G key. After earlyON minutes have elapsed, it will prepare and send the uplink comprising two bytes, the first one containing the last downlink received by the device and the second one being the coded value of the measured voltage of the 12V battery. When the uplink has been sent, the LMIC scheduler calls EV_TXCOMPLETE event which checks if a downlink has been received and if yes processes it. Once done, the code waits for prolongedON minutes before going to sleep until a time that is calculated by `gotosleep()` function by removing from next “round time” a number of minutes equal to earlyON.

March 2023 update:

It appears that the battery that I am using with the two PV panels charging it is too small capacity and the system runs out of power after a couple of days w/o sun. When sun comes back, the gateway is permanently powered up and in many instances, the consumption is too high and the system goes off again.

In order to solve this problem I have implemented a mechanism to properly switch OFF and ON the gateway based on measured battery voltage. Two additional parameters are defined in `parameters.h` file (`lib/ardbeescale` directory):

- `VOLTAGE_STOP_ALL` which is the battery voltage value below which the gateway won't be powered on any more (11.8V in current set-up)

- `VOLTAGE_RESUME` which is the battery voltage value that must be reached before gateway powering resumes

When the microcontroller is switched on, the set-up routine measures the battery voltage; if it is greater than `VOLTAGE_STOP_ALL` the gateway will be powered when needed. The global variable `BLOCK_POWER` is set to `False`.

Before switching on the Gateway, the main routine also measures the battery voltage and update the `BLOCK_POWER` flag.

- If `BLOCK_POWER` is `False` and voltage is greater than `VOLTAGE_STOP_ALL`, `BLOCK_POWER` remains `False`
- If `BLOCK_POWER` is `false` and voltage is lower than `VOLTAGE_STOP_ALL`, then `BLOCK_POWER` variable is switched to `True` and as a consequence the gateway won't be switched on
- If `BLOCK_POWER` is `True` and voltage is lower than `VOLTAGE_RESUME`, then `BLOCK_POWER` is maintained to `True`
- If `BLOCK_POWER` is `True` and voltage is greater than `VOLTAGE_RESUME`, then `BLOCK_POWER` is switched to `False`

Then the gateway is switched on if and only if `BLOCK_POWER` is `False`

10.4 Uplinks sent by this energy saving device:

`uplink[0]` is equal to the last downlink received. If this downlink was several bytes long, this will be `lastDownlink[1]`

`uplink[1]` is equal to coded value of the battery voltage, which is obtained by removing 768 from the 10 bits ADC readout. Knowing we have divided the battery voltage by 4, it means that a raw reading of 1024 corresponds to a voltage of 13.2V (4 times 3.3V).

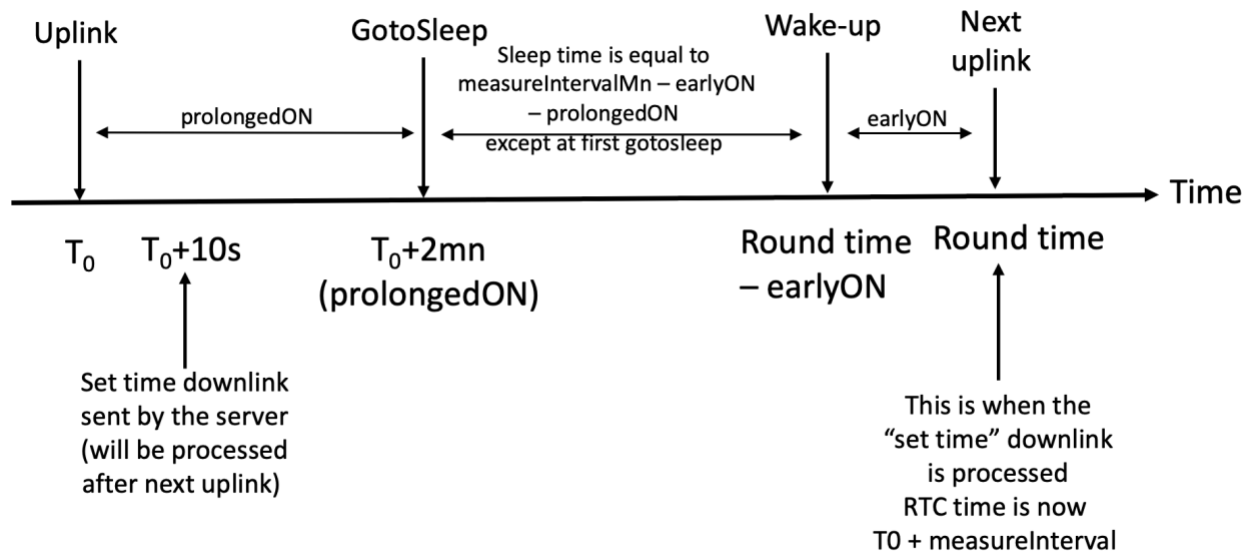
Therefore the formula to get the battery voltage from the coded value is:

$$batteryVoltage = (downlink[1] + 768) \times \frac{13.2}{1024}$$

10.5 Setting up the time of the energy saver unit.

The process for the scales is explained in section [2.1.9 Main Sketch;](#).

In the case of this energy saving device, the mechanism is a little bit different because the device will go to sleep not at "round time" (xH00, xH15, xH30, xH45 if measure interval if 15mn) but at round time minus earlyON minutes. The precise sequence of events is represented on following chart:



But in the end, there is still the same difference between two consecutive uplinks and therefore the code is the same as the one for setting up the time of the scales.

However, there is one caveat, which applied both to this energy saving device and to the scales. When the device is powered up, it does a bunch of different things to initialize the LoRa network and therefore the first uplink on the chart here above comes after a time that is largely undefined but different from 0 (the initial time of the real time clock at power-up). At **gotosleep()** time, the **gotosleep()** routine calculated the wake-up time to that it is a round time (minus **earlyON** in the case of the energy saving device). The time can be significant later that **measureInterval** minutes after the uplink has been sent and in this case, the real time clock will be wrongly set. In order to avoid this, there is only one solution: wait until the second cycle, where the time between two consecutive uplinks will actually be **measureInterval** minutes.

In other words: in all cases, do not set the time right after power-up. Wait until the next cycle.

10.6 Making sure everything is in synch...

The microcontroller real time clock may slightly drift over time, which could possibly result in a desynchronization of one or several devices versus the energy-saving device, which may prevent a correct transmission of uplinks and downlinks. In order to avoid this, several lines have been added in the crontab file of the raspberry pi running the back-end stack, in order to re-synch all the devices with TTN time once every day.

Here are the lines that have been added (command `crontab -e` to edit the crontab file):

Reminder: crontab line format is the following:

```
# mm hh jj MMM JJJ [user] task

# .----- Minute (0 - 59)
# | .----- Hour (0 - 23)
# | | .----- day (1 - 31)
# | | | .----- Month (1 - 12) OR jan,feb,mar,apr ...
# | | | | .---- Day of the week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
```

```
# | | | |
```

```
# * * * * * [user] command to launch
```

- *: All [days, minutes, etc]
- 1-3: select units 1 to 3 - {1, 2, 3}
- 1,4: select units 1 and 4 - {1, 4}
- */15: repeat every 15 units (every 15 minutes, 15 hours, etc)

Example:

```
05 00 * * * sudo python /home/pi/Documents/pythonScripts/getTime.py hso-energysaver 15
05 00 * * * sudo python /home/pi/Documents/pythonScripts/setTime.py hso-energysaver
```

Crontab remain silent when something goes wrong except is an email address is specified to which the error messages could be send (add MAILTO=your_email_adress at the top of the crontab file). Also make sure the pi is correctly set-up to send email. See [this post](#).

11 - LIST OF FILES

The project includes a number of different files which are listed here below and which are included in the GitHub repo containing this project code-named `ardbeescale-for-dummies`:

PlatformIO environment:

- `main.cpp` main source file
- `lib` directory of PlatformIO contains two sub direct directories:
 - `ardbeescale`: contains `hive-parameters.h`, which defines all the parameters that are specific from a scale
 - `keyfile`: contains `lorawan-keys.h` which contains the LoRaWAN node keys
- Following libraries have to be loaded (they are called by the main program):


```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
#include "DHT.h"
#include <RTCZero.h>
#include "HX711.h"
#include "hive-parameters.h"
#include <OneWire.h>
#include <DallasTemperature.h>
```
- `getTime.py` python script needed to set the time of a node. Explanation: [2.1.9](#)
- `setTime.py` python script needed to set the time of a node. Explanation: [2.1.9](#)
- `TTN-decode.js`: javascript file to be loaded on TTN console to decode the uplinks coming from the nodes
- `MQTT-pushDownlink.py` python script sending a downlink to a node using MQTT
- Blynk app related programs:
 - `blynkMain.py`: python script interfacing with blynk app
 - `blynkFunction.py`: python library used by `blynkMain.py` to send downlinks to the scales
 - `getStoreUplinks.py`: python script receiving the payload from the MQTT server, decoding it into individual values and storing those values in a file that `blynkMain.py` will retrieve and send to the smartphone app
 - `ardConf.py`: defines some key parameters for `blynkMain.py` scripts and blynk app to work. This is where the contents of the downlinks are defined, in particular those defining the time during which each sirup pump is turned on when the downlink triggering pump is received

Google sheet integration files:

- **gsheet file:** `ardbeescale-af01`
- `getDataFromUplink.py`: fetches the uplinks from the MQTT broker and stores corresponding uplinks in a directory
- `storedata.py`: store the payload retrieved by `getDataFromUplink.py` and store it in the google sheet file
- `config_ardbeescale01.py`: configuration file

PCB layout (Eagle) files:

- `plugFeather.brd`
- `plugFeather.sch`

12 - SETTING-UP THE SYSTEM...

The microcontroller software has to be loaded using PlatformIO and the ad hoc files provided on GitHub with all the needed libraries. Two files need to be updated:

- **hive-parameters.h** (in `lib/ardbeescale` directory). The last section of this file (labelled “scale parameters” has to be updated with the scale specific parameters, which are
 - `LOADCELL_OFFSET`: which is the ADC read out value when no weight is applied)
 - `LOADCELL_DIVIDER` which the number by which the ADC readout is changing when applied weight varies by 1g
 - `WEIGHTSLOPE`: which is the number of grams by which the scale measurement changes when loadcell temperature changes by 1 degree C.
 - `TIMECONSTANT` as defined in section [Weight temperature compensation](#) of this document
- **lorawan-keys.h** which, as its name suggests must be updated with the device specific LoraWAN parameters

Blynk application needs two python scripts to run:

- `getStoreUplinks.py`, which, as its name suggests, listens to the MQTT broker to get the uplinks that the nodes are sending and store them in a file that `blynkMain.py` will read
- `blynkMain.py`, which is the script getting the uplinks information from the files that `getStoreUplinks.py` has created and sending corresponding data to the Blynk app. It's also the script that sends downlinks to the nodes, based on what the user does with the app.

For some reasons, the blynk library is not 100% stable and sometimes crashes. There is no damage with the crash but simply the program stops. In order to cope with this I have set-up `blynkMain.py` as a service (with auto restart option) using `systemd`. For starting a script as a service, one need to create a file `servicename.service` in `/etc/systemd/system`. In our case, we will launch two services:

- `blynkMain.service`
- `getStoreUplinks.service`

The content of these files is the following:

```
[Unit]
Description= Launch python script blynkMain.py
[Service]
ExecStart=/usr/bin/python3 /home/pi/Documents/pythonScripts/blynkMain.py
```

```
Restart=always (the script will be restarted if it crashes)
SyslogIdentifier=blynkMain
User=pi (remove this line to launch the script as root)
Group=pi (remove this line to launch the script as root)
[Install]
WantedBy=multi-user.target
```

Once the file is created,

- `systemctl start service_name` to launch it
- `systemctl stop service_name` to stop it
- `systemctl status service_name` to check its status
- `systemctl enable service_name` to validate auto launch at start-up
- `systemctl disable service_name` to stop auto launch

To check the outputs of the script, the command is:

`journalctl -u service_name`. (`journalctl` manages the max size of the log file). Up and down arrows are used to scroll through the pages of the log file. Q to quit.

To visualize the last 100 lines of the log file : `journalctl -u service_name -n 100`

In the latest version of `systemd`, it is possible to redirect the outputs to a specified file but the Raspbian version that I am using is still not updated with this new feature.