POLYTECHNIC SCHOOL OF ENGINEERING OF GIJÓN.

DEGREE IN ELECTRONIC AND AUTOMATIC INDUSTRIAL ENGINEERING

ROBOTICS ENGINEERING AREA

APLICACIONES DE LA TECNOLOGÍA DE COMUNICACIÓN LORA EN ENTORNOS RURALES

APPLICATIONS OF THE LORA COMMUNICATION TECHNOLOGY IN RURAL ENVIRONMENTS

STUDENT: D. ALEMANY SUÁREZ, Álvaro
TUTOR: D. RUBIO GARCÍA, Ramón

DATE: JULY 2022

Index.

Index of Figures

1. Introduction and State of the Art

1.1. Summary

Cattle in Asturias are prone to wildlife attacks such as wolves and bears, resulting in a high economic cost for the farming community. It is possible to receive money in compensation, however one must prove that the animal has died due to an attack, and not of natural causes or other accidents.

In order to determine the cause of death, time is of essence. On many occasions, a farmer only finds a few bones weeks after the incident, and in such cases it would be impossible to ascertain the cause of death. Even when a well-preserved corpse is found and wolves have been on scene, there is no guarantee that the cattle has been attacked, as wolves are not only hunters but also carrion feeders. Specialists must arrive at the scene as soon as possible.

There are several commercial GPS-based geolocation systems available, but what most have in common is their high cost. A farmer is not willing to spend thousands of euros for possible compensations of a few hundreds.

The goal of this project is to study the use of a new geolocating system designed for cattle in the Principality of Asturias. This system will use the Long Range (LoRa) technology for data transmission, which allows data transfer in the range of dozens of kilometers, albeit with a very low bandwidth. This data limitation is not a big problem, though, as we only need to send a few bytes of information with the coordinates.

The key features of this system will be a low battery usage, a low-time response after the death of the cattle head, and the ease to replicate the system. Further details on the device will be discussed in the next sections.

In this first section, we will discuss the goals for this project, the current state of the art and how different geolocation systems can be used to monitor cattle.

In the second section of the document, Concept Design, we will attempt to describe in broad strokes how a similar device to this project could be replicated, and what the requisites may be. The goal of this section is to allow anyone in the future to build a similar device or improve on our prototype, regardless of the availability of specific components.

In the third section, Detailed Design, we will discuss all the technicalities of the selected devices and how to implement them in a working prototype. Whereas the Concept design may be red as "what does one need to make something like this?", the Detailed design has a different approach: "how exactly did we build our prototype?".

1.2. Objectives

The goal of this project will be to study the use of the LoRa technology for cattle geolocation in Asturias. There are already commercial uses of this technology, but in this project a new prototype will be developed. The requirements for this prototype will be:

- The device should be compact enough to fit on the collar used for cattle such as cows, horses, or goats. The exact location is irrelevant as long as the animal can carry it comfortably.
- The prototype should be made of a single package, without any cables or moving parts, so an animal cannot rip them off with their horns or by getting it stuck on external obstacles such as branches.
- The prototype should have a large lifespan, so as not to require periodic battery changes. It would be a minor issue for a single animal, but not optimal for a large herd.
- The device must fulfill three tasks:
    - Measure the activity of the animal using sensors.
    - Monitor the precise location of the animal.
    - Communicate these data when needed via LoRa.

A way of implementing a cost-effective solution would be to reduce the number of costly components. With a simple prototype that simply sends the location periodically would make the device much cheaper, smaller and a battery lifespan that would be drastically increased. A lower price and longer battery duration would be a good alternative for a farmer who is just interested in the location of their cattle.

Another issue for smaller animals is the size of the collar. The weight could be a limiting factor for a prototype, especially when the device is designed to be worn by an animal for the rest of its life.

Figure 1 - A goat wearing a tracking device

1.3. State of the art

The use of geolocation technologies has seen an important increase in the last years, with several startups arising from the latest developments in the so-called "Internet of Things".

"The use of geolocation will become more relevant in the farming sector by the day. Low power monitoring is expected to reach 21.000 million euros in 2025." (1)

There are many reasons why farmers may want to implement geolocation solutions for their cattle. The most frequent ones are detailed here:

- Geofencing
- Monitoring the general location through a small sample of individuals
- Monitoring health issues in the cattle using a variety of sensors.
- Interpreting received data to infer behavioral changes. For instance, with an accelerometer, you may detect whether an animal is resting, walking, or running.
- Identifying wildlife attacks by correlating abnormal data, such as an activity spike during nighttime hours.

1.3.1. Geofencing

A livestock farmer may have a controlled area, such as their own private fields in which only his cattle roams. Geofencing implies delimiting a perimeter, a "virtual fence" of sorts. If an animal with a geolocation device crosses that virtual boundary, the farmer receives some kind of notification, which may contain information about the device's location.

For instance, the device could send the information through the network and the farmer would receive an SMS or an email with the point of breach of the geofence.

Another use of geofencing with a more active approach would be a device designed to teach the animal not to get away from the perimeter. Some methods include delivering a small electric shock to the animal, not to harm it but to scare it. Other alternatives include the use of audio warnings that have the same function: scaring the animal and making it stay inside the virtual fence.

A mixed approach would be "teaching" the animal what those sounds mean: making a noise when the animal gets away from the geofence, and a small shock when it deviates further, or when it stays outside the designed area for too long.

Some companies such as Nofence (3) implement this last method and have their own app to monitor in real time the location of their cattle, with already 59,000 heads of cattle from 3600 farmers using their devices. Their app has a map functionality in which you can directly draw to define the area of the pasture.

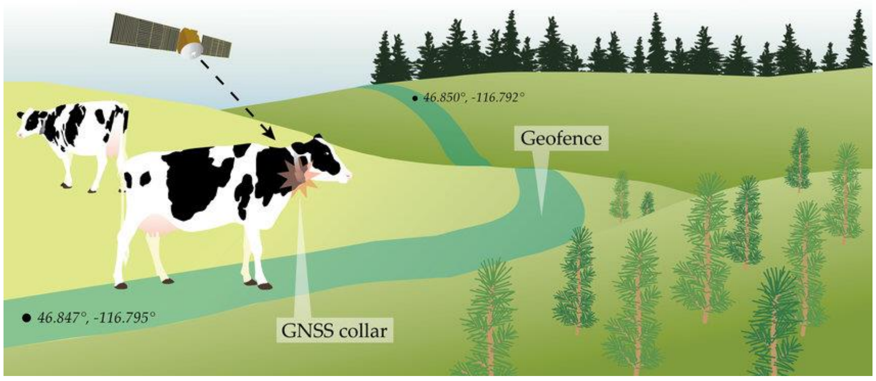
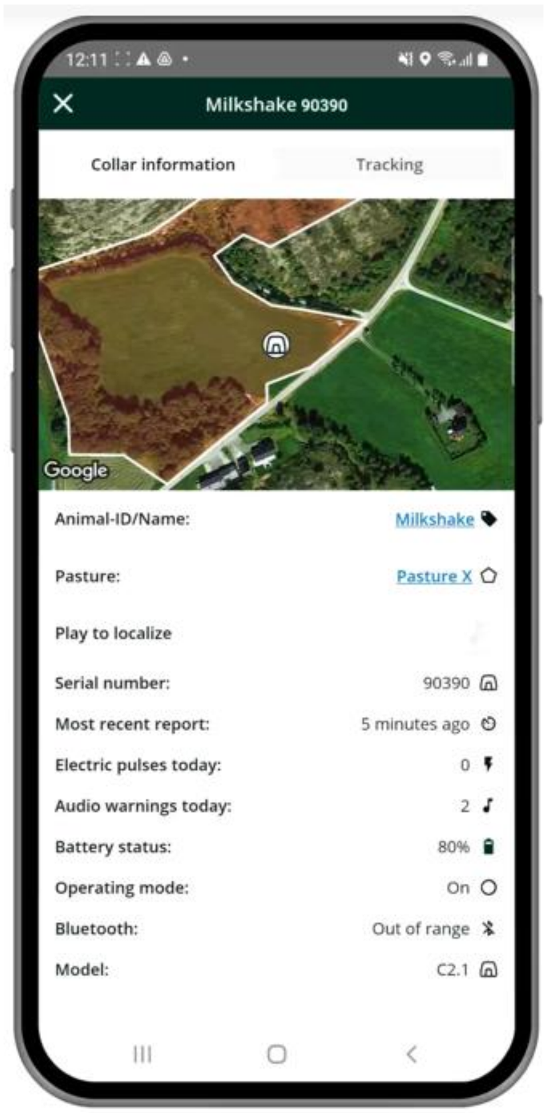


Figure 2 – a geofencing system


12:11
Milkshake 90390

Collar information | Tracking

Animal-ID/Name:	Milkshake
Pasture:	Pasture X
Play to localize	
Serial number:	90390
Most recent report:	5 minutes ago
Electric pulses today:	0
Audio warnings today:	2
Battery status:	80%
Operating mode:	On
Bluetooth:	Out of range
Model:	C2.1

Figure 3 – Screenshot from the Nofence website showing their application.

Other examples of geofencing are designed to prevent cattle theft. IndiaNIC (4) is an Indian company working on this issue, and they declare that 70% of livestock thefts are not reported. Although livestock theft is not in the scope of this project, it is a relevant issue to study when developing this kind of applications. In their case, their tags don't interact with the animal at all, only show their location. This would not be an issue of an animal wandering away from a designed pasture and having to redirect it, but a theft in which the animal cannot do anything about their position. In this case, it would be unfair to "punish" the animal either by shocks or audio cues, which would stress the cattle. Additionally, if the animal was further distressed, it could be a sign for the thief that something is off, and it could discover the presence of the GPS tag.

Another interesting application for geofencing stems from the option of constantly redefining the virtual fence. A farmer that periodically changes "allowed" sectors of their farm can optimize the grazing patterns of their livestock. This can be also applied to free range cattle that depend on transhumance, moving to different pastures from season to season. With the transformation of the rural world and the decline of traditional transhumance farming, the upkeep of hills and mountains is an important issue that needs to be addressed by regional and national authorities. Defining and optimizing grazing regimes is a key factor not only for the sustainability of cattle production systems, but also for preserving the biodiversity of the environment. (5)

1.3.2. Monitoring the general location through a small sample

This is a common use of geolocation for cattle. One of the main problems that livestock farmers encounter is the cost of the devices. In many cases, the solution is to buy a limited amount of geolocation devices and use them on a small sample of the herd. Assuming animals tend to stay together, by monitoring a bunch of individuals can give a somewhat accurate estimation of their location.

However, the advantage of reducing the cost of this solution comes with certain drawbacks. The most obvious one is that not all of the individuals are located and so, if there is any stray animal, detecting its absence is pure coincidence. Any animal without a location device is pretty much lost.

Figure 4 - Cow wearing a Digitanimal collar.

Perhaps monitoring every single livestock head is overdesigning for certain farmers, certainly with the prices that some devices present. For instance, if a geolocation solution implies a cost of 200€ for each device and the farmer receives only 100€ of compensation for a dead animal, tagging every single animal does not sound very appealing to the farmer. Even with the benefits of having the whole herd identified, the initial cost may be too steep.

The amount of money received in compensation for a wildlife attack on cattle varies with the type of animal affected. In 2018, the economic compensation for different cattle (2) was in average:

- 323 € for horses
- 509 € for cows
- 78.40 € for sheep
- 102 € for goats

There are, however, instances in which monitoring the whole herd would be interesting, described in the following points.

1.3.3. Monitoring health issues in the cattle using a variety of sensors.

There are many kinds of sensors used to monitor the health of the animal. There are certain ethical and practical questions that arise from different practices. For instance, some temperature sensors can be implanted as a subdermal chip, which is a very minor surgery that can be performed without anesthetics. However, it is understandable that this is not a desired procedure for many livestock farmers, especially with those dedicated to meat production, as the issue of how to recover the devices after the sacrifice might be an inconvenience.

Many devices present a single package located somewhere on the collar of the cattle, with all the sensors present in a single place. Other solutions have been studied, where for instance a pulsimeter has been placed in the ear of a cow and the rest of the sensors located in the collar. However, animals tend to destroy any kind of cable present in those solutions when left alone. For applications where sensors are momentarily placed for veterinary supervision this is not a big issue. However, when we think about devices supposed to stay on the animal permanently, this is not a convenient solution.

1.3.4. Interpreting received data to infer behavioral changes.

For instance, with an accelerometer, you may detect whether an animal is resting, walking, or running. Different behavior patterns can be identified by a single accelerometer.

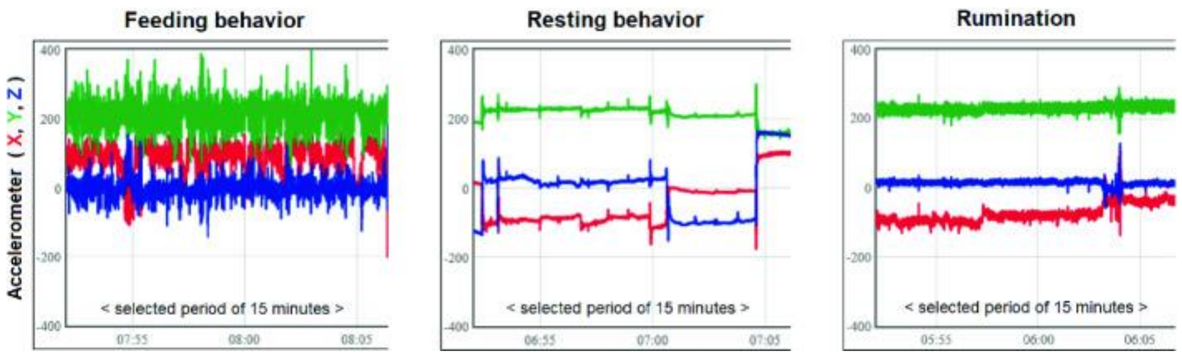


Figure 5 - Different activity levels used to identify behavior (6)

The variation of these graphs can be even studied during a longer period of time to identify the moment in which the animal enters heat. The average activity levels obtained from the accelerometer can be used to determine when a cow increases its activity during heat. Another variable to study this case would be the decrease of rumination time, which also correlates to the animal being in heat. (6)

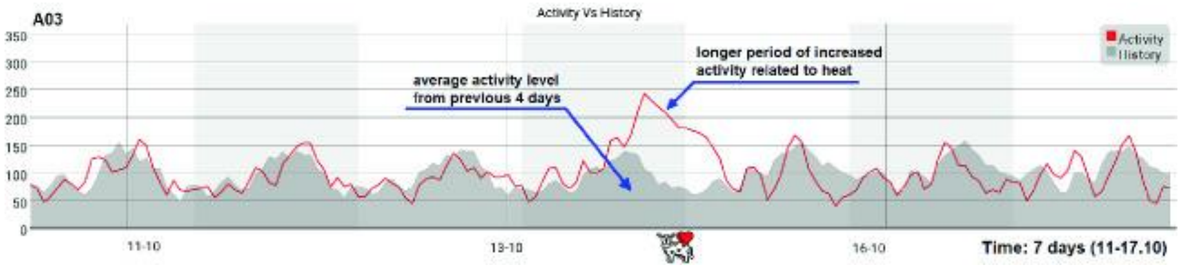


Figure 6 - Increase of activity during heat

Another example of possible sensors may include pulsimeters, which are generally located in the ear of the animal, as they depend on a light emitter – sensor pair to function, and the ear of the animal is thin enough to allow measuring the blood flow. However, placing devices on an animal's ears can be cumbersome. The pulsimeter is small and light but connecting it to a microcontroller and adding a telecommunication device gravely increases the weight of the product. According to Cowmanager (7), when a cow gets an infection, blood of less important areas such as the ears moves to its vital organs. The decrease of pulse strength can inform us about a possible diseased individual.

A thermometer can also be used to monitor vitals of the animal. However, for cattle, the most common place to place a thermometer is next to the anus, at the base of the tail. Other possibilities may include placing it directly at the neck of the animal, exposing part of the device. However, collars tend to be loose, and further studies and calibration is needed.

1.3.5. Identifying wildlife attacks by correlating abnormal data.

There are organisms dedicated to tagging predators such as wolves, monitoring their location and studying their behavior. An example of tagging predators is the study made in Namibia (8), in which over a hundred cheetahs were tagged and their behavior analyzed. The result was a drastic decrease in mortality of certain herds that avoided specific areas, with a result of an 86% reduction of cattle mortality.

However, not all wild individuals are tagged, so despite the invaluable information they provide, it is definitely not complete.. Additionally, the data provided by these wildlife studies is logically not available to the public, as it contains sensitive information related to protected species. If everyone could know where a wolf pack is located, it would be extremely easy for any hunter or poacher to obtain this information.

Tagging cattle may offer a complementary source of information that does not involve releasing sensitive information. In a similar way to accelerometer data used to monitor heat of an individual, if a large number of animals gets an activity spike during the night, this may show an instance of an attack by a wild predator such as a wolf or a bear. This information is solely dependent on the cattle's activity instead of tracked predators. For the farmer, the information received is similar: "your animals are startled by something, possibly a predator" instead of "there are wolves in your area". The result is the same: the farmer approaches the herd afterwards and checks for any signs of an attack, but without any risk for the life of the predator.

As mentioned before, there are other commercial systems for cattle geolocation. Among the best-known systems are Tellus GPS for wildlife and DigitAnimal for cattle. While these are already established companies with great results based on their reviews, one of the main problem that farmers encounter is the high price of the devices. How are they going to buy a collar costing over 100€ to protect a sheep, if the money they might get in return is 80€? (1) Obviously, these numbers differ a lot depending on the type of cattle we focus on, but the argument is similar for all of them.

An interesting way to implement this would be to change the working principle of the device. Instead of having a system that constantly broadcasts the location of the animal, the goal would be monitoring the vitals of the animal, and when there are signs that said animal has died, then a notification is emitted with the current location of the corpse and the last time vital constants were found.

Additionally, the data will be transmitted using the LoRa technology. At the moment there are very few LoRa antennae available in Asturias, but the Principality of Asturias plans to extend the connectivity throughout the whole region using the already existing TDT towers of the region, around 170 antennae present in the Principality. (10). The solution would be installing a LoRaWAN antenna in the vicinity of the farm, offering a service area of around 10 – 15 km (as long as there is a direct line of sight).

One big problem of the region is the complicated orography that it presents. Line of sight is not usually an issue regarding satellite technology, but direct line of sight to a LoRa antenna, located in the ground, is not guaranteed when grazing in Asturian terrain.

The key aspects of using the LoRa technology are its long range, reaching multiple kilometers, the low power consumption of the LoRa devices, lasting years on a battery and the relative low cost of the nodes, but at the cost of a low bandwidth.

2. Concept Design

In this section we will analyze the requirements of the project, explaining the parameters our components must meet, study the different options that fulfill those prerequisites, and discuss what solution will be implemented in our prototype.

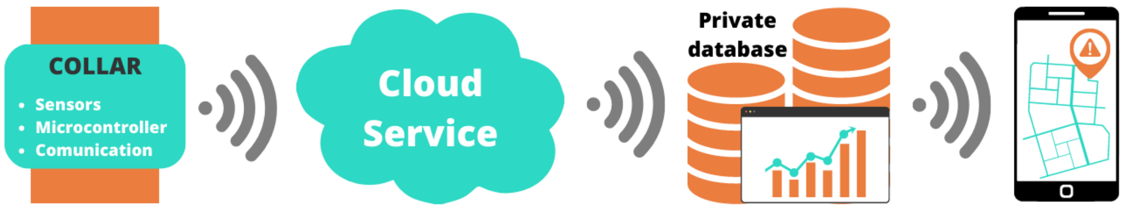


Figure 7 - Diagram showing the different parts of the project.

The project is a combination of several parts:

- A device located on the animal, which we will refer to as "collar", which contains the required sensors, microcontroller, and communication device.
- A way to extract the data from the device and send it to our database.
- A server on which we can store our data plus some visualization tools.
- A way to access the data and send alerts in case something goes wrong.

2.1. Device for the collar

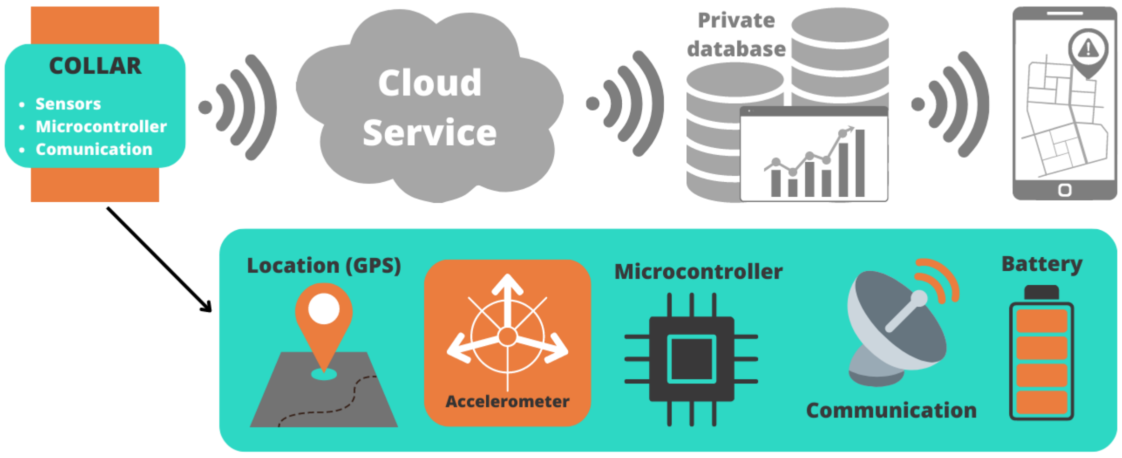


Figure 8 - Parts of the collar

The device located on the collar works by integrating five parts: a device to gather location information, a way of detecting the animal's vitals, way to communicate that information, a microcontroller that manages all the data and communications, and a power source.

These need not be five separate parts, as there are several commercial options that integrate two or more of these devices.

2.1.1. Location

Among the different geolocation methods, GPS is the most popular, and stands for Global Positioning System. This system is formed by a constellation of satellites operated by the USA and works together with other constellations in a broader system called GNSS (Global Navigation Satellite Systems). In this document we will refer to GPS and GNSS indistinctly, as GPS is the commonplace name of the system.

GPS receivers work by "listening" to the signals constantly emitted by the constellation, receiving several signals from different satellites, and using the information within to calculate location and time. If we manage to receive data from four different satellites or more, then we will be able to precisely calculate our location using trilateration. Most GPS units have a precision between 10 and 20 meters, which should be more than enough for locating a relatively large animal such as a cow in an open field.

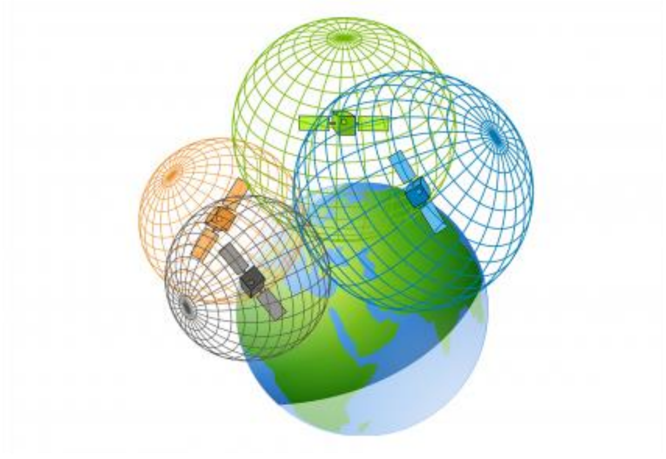


Figure 9 - Trilateration of 4 satellites

These are the most common factors that may decrease the positioning accuracy in a GPS device (15):

- The blockage of a signal due to structures such as trees or buildings in the way.
- The use of the device indoors or underground.
- The reflection of the satellite signals bouncing from buildings or walls.

These are factors that derive from the use of the device in inadequate conditions. Problems may arise when developing and testing the device indoors or in the middle of a city, but there should not be any related issue when testing the GPS system in an open field, which is conveniently the intended use of the device.

Outside factors include radio interference, temporary satellite maintenance, or solar storms. However, there is nothing we can do about those issues and must simply hope for the best.

NMEA sentences

GPS signals are received as an ASCII string using different formats. Developed by the National Marine Electronics Association, and therefore called NMEA sentences (17). There are many kinds of sentences containing different data, the following is an example:

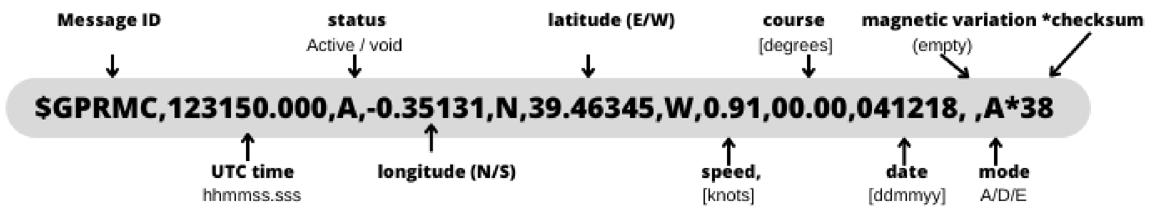


Figure 10 - Deconstruction of a NMEA sentence

"**A**", which stands for "active" means the location is fixed and correct. If an invalid location result is given, the signal would send "V" (void) instead.
-0.35131 is the longitude of the device, and **N** stands for North.
39.46345 is the latitude of the device, and **W** means West.
0.91 is the speed, measured in knots.
00.00 represents orientation, measured in degrees.
041218 is the date (4th of December 2018).

There are many software libraries developed to extract the required data from these kind of sentences, so for our project we simply need a GPS unit compatible with the selected microcontroller. The device will always "listen" to the satellites and transmit the received sentences to the microcontroller. Most GPS devices use only two pins of a microcontroller to transmit the data via UART (assuming they are connected to the same Ground pin).

2.1.2. Possible sensors

Detecting pulse:

This would probably be the most straightforward way of detecting the vitals of the cattle, but it is impractical for out prototype, for several reasons:

Many pulsimeters consist of a pair of devices: a light emitter and a photo receptor. The issue with these devices is how to attach it to the animal. For humans it is quite simple: a finger is placed between two pieces and the pulse is measured among other capabilities such as detecting oxygen rate in blood. Cows and other cattle do not have fingers to do this, and such we must find another place. The best option is to place them on the ears of the animal.

Figure 11 - Light emitter and photoreceptor pair

However, more problems arise from this solution mainly the fact that the prototype as a whole would be too heavy to hang from the ear of the animal. The pulsimeter should be separated from the other devices, having two options:

- Option 1: using cables connecting the two parts. Animals do not respond well to cables or any loose parts. Interviews with farmers and animal healthcare associations have taught us that part. In their example, they wanted to monitor the heart rate of a cow on its way to the slaughterhouse, using a belt designed especially for that purpose. At the end of the trip, other cows had used their horns to free the target from the belt, seeing it as a foreign object that was causing distress.
- Option 2: not using cables. For this, we would have to add another microcontroller, communication system and power source to send the data to the main one on the collar. This will increase price, complexity of the design, and once again weight of the device hanging from the animal's ear.

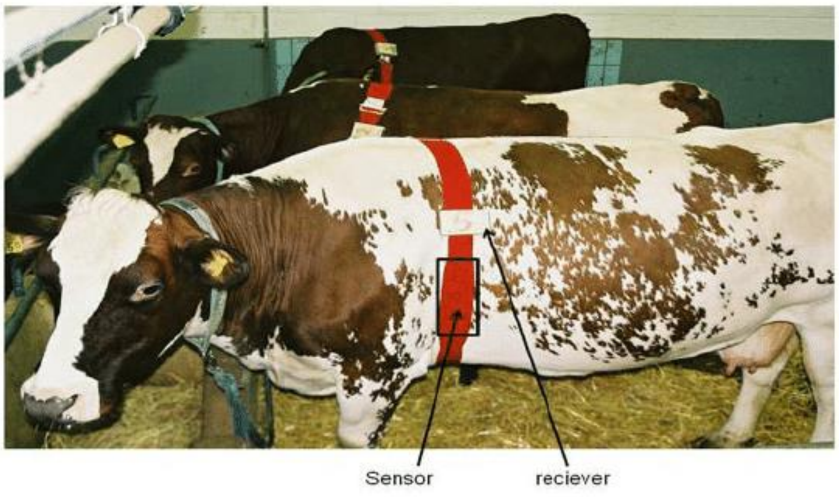


Figure 12 - A cow with a belt and sensors to measure heart rate (18)

We have also found another alternative cardiac meter, this time packaged in a single device. However, the cost of this device rises to around 50 €, so it is not compatible with our goal of making the device affordable.

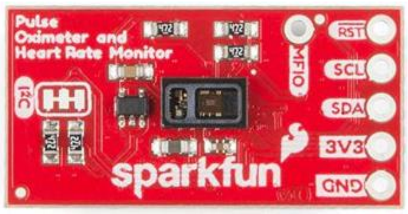


Figure 13 - Single-part pulsimeter device (19)

For these reasons, we have decided not to use a pulsimeter to monitor the vitals.

Detecting temperature

The temperature of an animal would be another way to indicate its death, as the corpse cools down after its demise. Temperature sensors are not uncommon in electronics and their price is usually not very high.

Based on temperature alone, however, it would be difficult to detect the state of the animal. The temperature sensor would have to be on its skin, if not under (even having a subdermal implant for optimal results, which we do not desire to use). Moreover, factors such as ambient temperature, sunlight incidence or heat from the other devices may alter the result. We believe that temperature may be a useful piece of data to acquire, but not the only factor that should be considered for determining the state of the animal.

Detecting movement:

- **Working principle of an accelerometer**

Accelerometers have a mechanism inside them that works like a mass-resort system, allowing the measurement of acceleration in terms of the displacement of the moving parts. Such mechanism has the following parts:

- A stationary frame that links the accelerometer to the object we want to measure.
- A movable mass inside, called a "proof mass".
- A suspension system that holds the mass, usually simple springs.
- A damping system. Sometimes a specific damper is placed inside, other times air resistance suffices.
- A sensor that measures the displacement of the proof mass.

As the target moves, the fixed frame moves with it, but the proof mass has its own inertia that opposes such movement. However, the suspension system counteracts this movement and avoids the proof mass colliding with the frame. Then, the displacement sensor measures the distance between the mass and the frame.

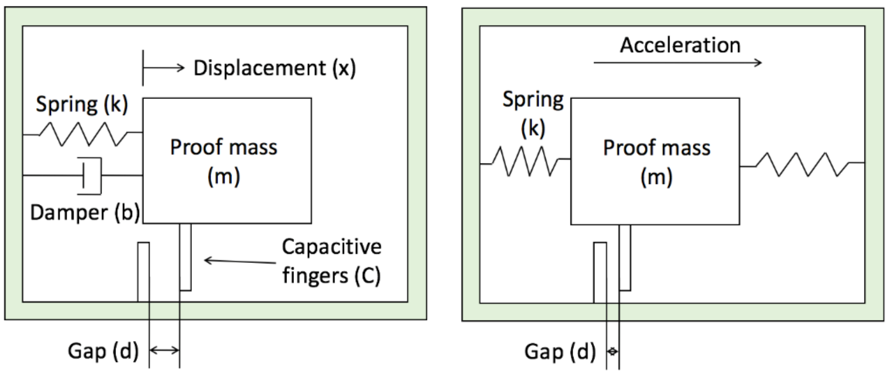


Figure 14 - Simple diagram for capacitance-based accelerometers

In modern accelerometer devices, the measurement of this distance is made using capacitance. The proof mass has a capacitor finger or plate attached, whereas the frame has another fixed capacitor finger or plate. The movement of the mass changes the distance between the plates, affecting the capacitance.

$$C = \frac{k\varepsilon_0 A}{d} \tag{4.1}$$

In equation 1, A is the Area of the plates, ε_0 = 8.854x10^{-12} F/m the permittivity of space and k the relative permittivity of the dielectric material between the plates. As the area is a construction parameter, and k is the permittivity of air (approximately 1), the numerator is a constant, meaning that the change in capacitance is inversely proportional to the distance between the gaps.

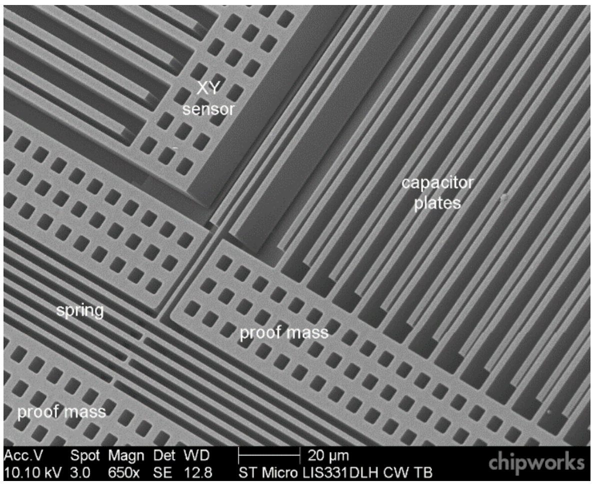


Figure 15 - Electron image of a common micro accelerometer

With another electronic circuit, the capacitance is converted into voltage and therefore can be easily read by other electronic devices. The accelerometer must be calibrated before being put into use, and one aspect to be considered is that, even though the frame is not moving, every object is going to be subjected to earth's gravitational acceleration.

- **Working principle of a gyroscope**

Microelectromechanical systems (MEMS) gyroscopes have become part of integrated circuits and are now available for many applications. A very simplified way of explaining these systems is that the vibrations of an object continue vibrating in the same plane even if the frame experiences a rotation. The Coriolis effect makes the vibrating object apply a force on the support structure, and the measurement of such forces allow the calculation of the rotation. (12)

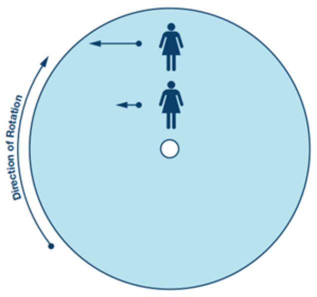


Figure 16 - Coriolis acceleration

If we have an object vibrating inwards and outwards of a disk that does not rotate, it will not experiment any Coriolis force. However, when the frame experiences a rotation, the Coriolis effect will exert a force in the direction of rotation while the object moves inwards and contrary to such direction when moving outwards.

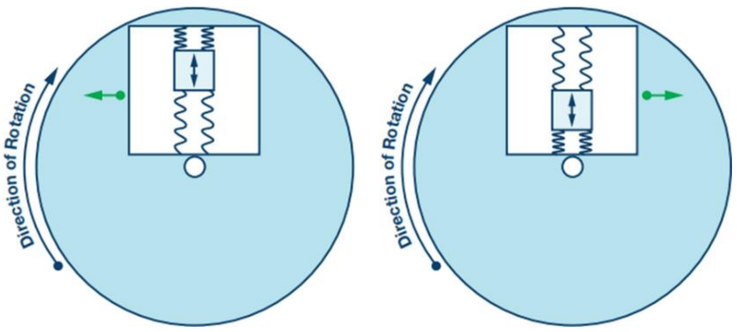


Figure 17 - Coriolis forces on a vibrating object rotating on a disk

We can place this structure inside a spring system with capacitive fingers, similar to the accelerometer discussed previously. The Coriolis force will make such system move laterally when the disk is rotating and staying still if no rotation is observed.

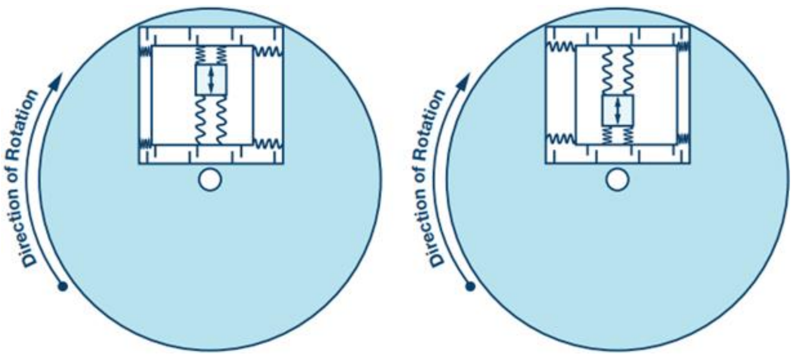


Figure 18 - Frame displaced laterally due to the Coriolis forces

If we have two proof masses (similar to the accelerometer system) vibrating in a plane that rotates with an angular velocity **ω_r** and the mass having a velocity **v** relative to its rotating reference frame, the Coriolis effect will make the masses experience an acceleration equal to:

$$a_c = 2(\omega_r \times v) \tag{4.2}$$

Using Newton's Second Law (F=ma), we can establish that the Coriolis acceleration will result in a force that depends on the proof mass **m_p**

$$F_c = 2m(\omega_r \times v) \tag{4.3}$$

The position of the proof masses vibrating in the plane can be expressed as a sinusoidal wave of amplitude **A_{ip}** (in plane) with the angular velocity of the vibrations **ω_v**

$$\mathrm{y} = A_{ip} \sin{(\omega_v\, t)} \tag{4.4}$$

Therefore, calculating the derivative gives us the in-plane velocity as

$$\mathrm{v} = A_{ip}\, \omega_v \cos{(\omega_v\, t)} \tag{4.5}$$

Similar to the accelerometer, the gyroscope will have a spring mechanism that will allow it to move with respect to the frame. For a spring, Hooke's Law relates the relationship between the force exerted on an object and the displacement **x** such object experiments, proportional to the spring constant **k** of the mechanism:

$$F = kx \tag{4.6}$$

Such spring constant will be estimated by the manufacturer and used to calibrate the gyroscope. We consider that the spring is out of plane with respect to the vibrating mass. With equations 4.3, 4.5 and 4.6, we can determine the out of plane displacement **x_{op}**:

$$x_{op} = \frac{F_c}{k} = \frac{2m(\omega_r \times v)}{k} = \frac{2m\, \omega_r\, A_{ip} \omega_v \cos(\omega_v\, t)}{k} \tag{4.7}$$

If we measure the displacement **x_{op}** we can calculate the rate of rotation **ω_r** perpendicular to the motion of the proof mass. Measuring the displacement is made the same way as we did with the accelerometer: the displacement affects the capacitance between the sensor fingers, and this capacitance is transformed into a voltage that we can read.

2.1.3. Microcontroller

None of the sensors would be able to work without a "brain", the microcontroller. A large number of microcontrollers exist with a great range of prices, available connections, processing power and memory storage.

Among the most popular devices is the brand Arduino, whose framework is the most widely used for developing this kind of product. This is the first option we thought of when deciding which microcontroller to use. The Pro Micro, Pro Mini and Nano series have a relatively small size but a large number of pins for the different connections required.

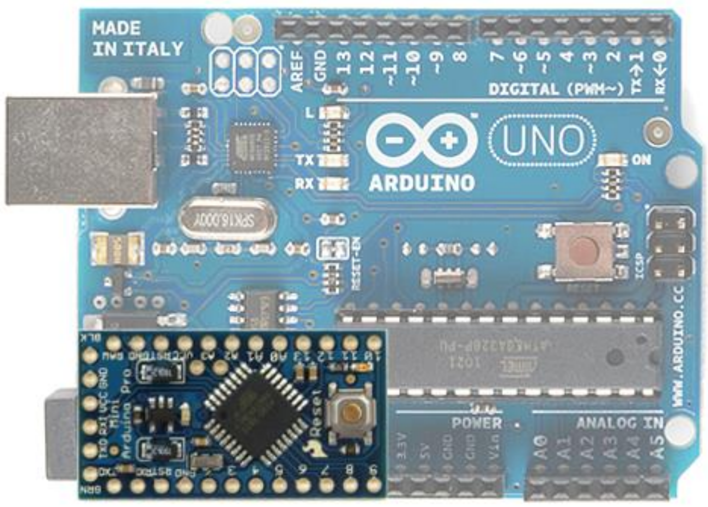


Figure 19 - Arduino Pro Mini compared to Arduino UNO

Initially we thought of an Arduino Pro Mini due to the reasons stated above. However, after testing all the individual sensors, we had to switch the microcontroller due to the available memory (2 kB SRAM and 32 kB Flash) not being sufficient for the final program, which occupies 5.3 kB of RAM and 87.7 kB of Flash memory.

With LoRaWAN communication being a specification of the project, we decided to use the Heltec Cubecell HTCC-AB01 development board, which already has LoRaWAN communication system integrated, which will result in a simpler device.

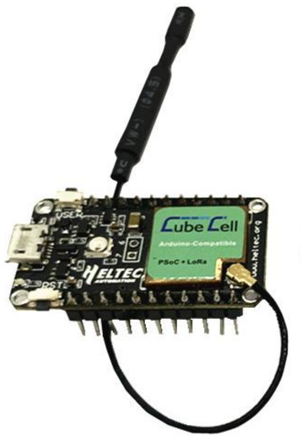


Figure 20 - Heltec Cubecell HTCC-AB01, including the LoRaWAN antenna

2.1.4. Communication

As mentioned previously, one of the requirements of the device is to use LoRa, which is a kind of wireless modulation based on a technology called Chirp Spread Spectrum (CSS). The word “Chirp” is also an acronym for “Compressed High Intensity Radar Pulse”. As the name indicates, it was originally developed for radar applications, but has now extended to other uses. This technology helps transmitting signals reaching very long ranges (distances vary but are typically in the order of tens of kilometers).

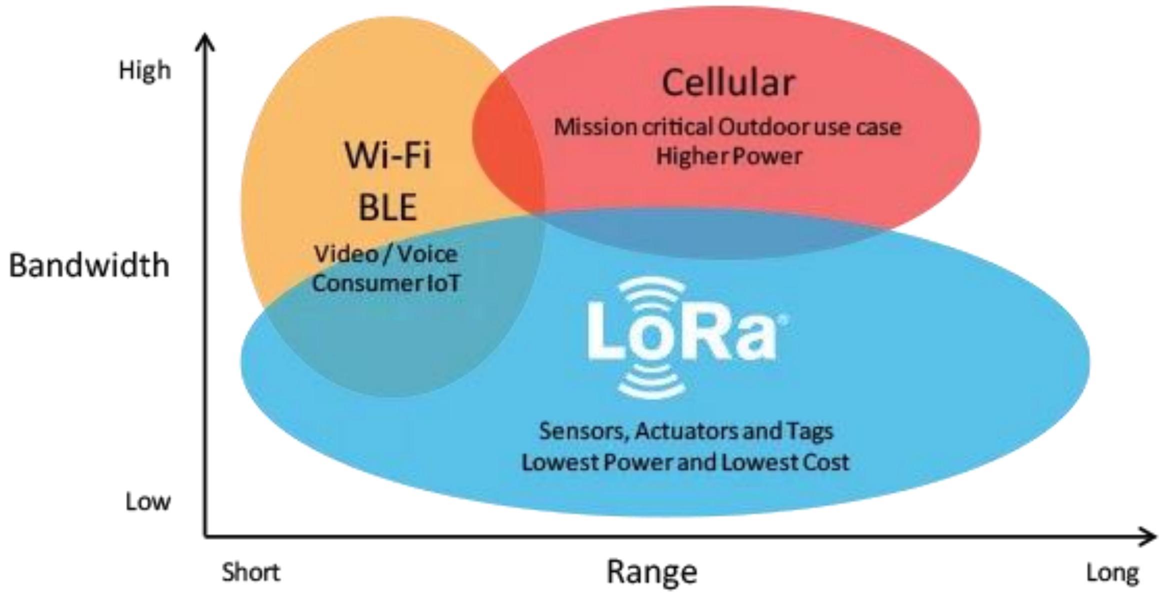

High
Bandwidth
Low
Wi-Fi
BLE
Video / Voice
Consumer IoT
Cellular
Mission critical Outdoor use case
Higher Power
LoRa
Sensors, Actuators and Tags
Lowest Power and Lowest Cost
Short
Range
Long

Figure 21 - Comparison between LoRa, WiFi and Cellular networks

Chirp modulation is a kind of frequency modulation in which the signal “slides” across two established frequencies. Moving from the low frequency to the high one is called “up-chirp” and the opposite is called “down-chirp”. This up-chirp or down-chirp shifts could substitute the digital “ones” and “zeros”.

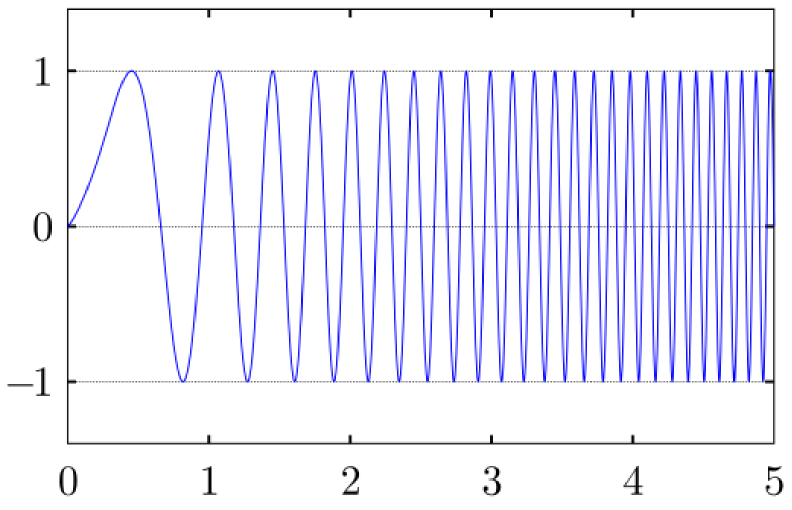


Figure 22 - An example of up-chirp in time domain

However, this is not exactly how the modulated signal works. In reality, there are different configurations for the signals, and one of the most important concepts is the Spreading Factor (SF). As a simplified explanation, the lower the SF, the lesser amount of data is sent, but the faster you can send your signals (higher data rate). However, sending faster rates also means reducing the power of the signal and the range of our transmission.

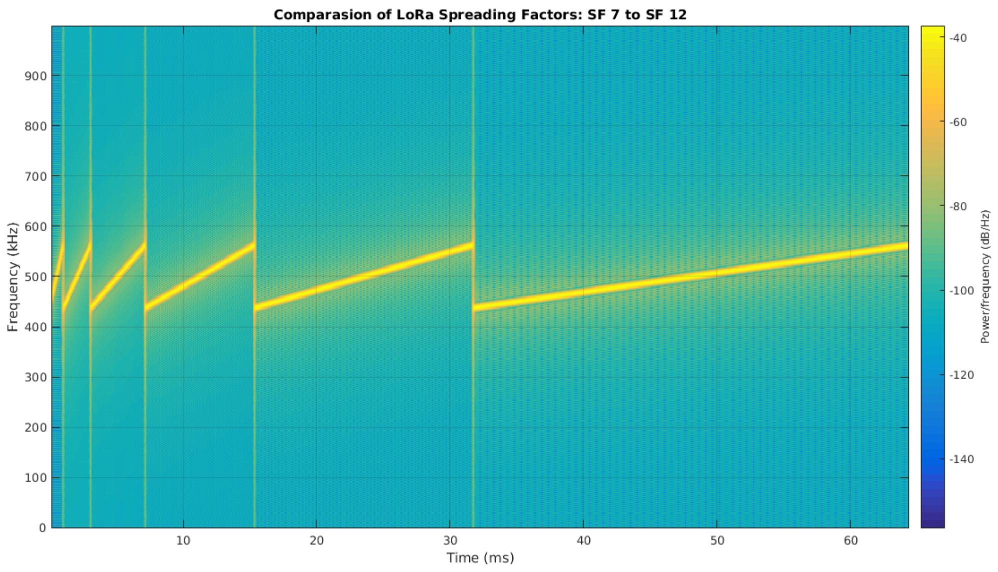


Figure 23 - Comparison of the Spreading Factors from SF7 (left) to SF12 (right)

LoRa uses 6 different Spreading Factors, named SF7 to SF12. Each part of the signal is divided into 2^{SF} sections called "chips", so a signal of SF7 has 2^7=128 chips, and a signal of SF8 has 256. This means that we can send a number of bits equal to the SF in each symbol of the payload. Increasing the SF by one means sending twice the number of bits and therefore taking twice as long.

The receivers are always listening for new messages. The preamble announces the arrival of a new message and consists of eight up-chirp symbols. It is followed by two down-chirp symbols that are used for precise time synchronization, and only then is the payload delivered. For instance, if we configure the payload to have 5 packets of data, the signal will have 15 symbols: 8 for the preamble, 2 for the synchronization and 5 for the payload.

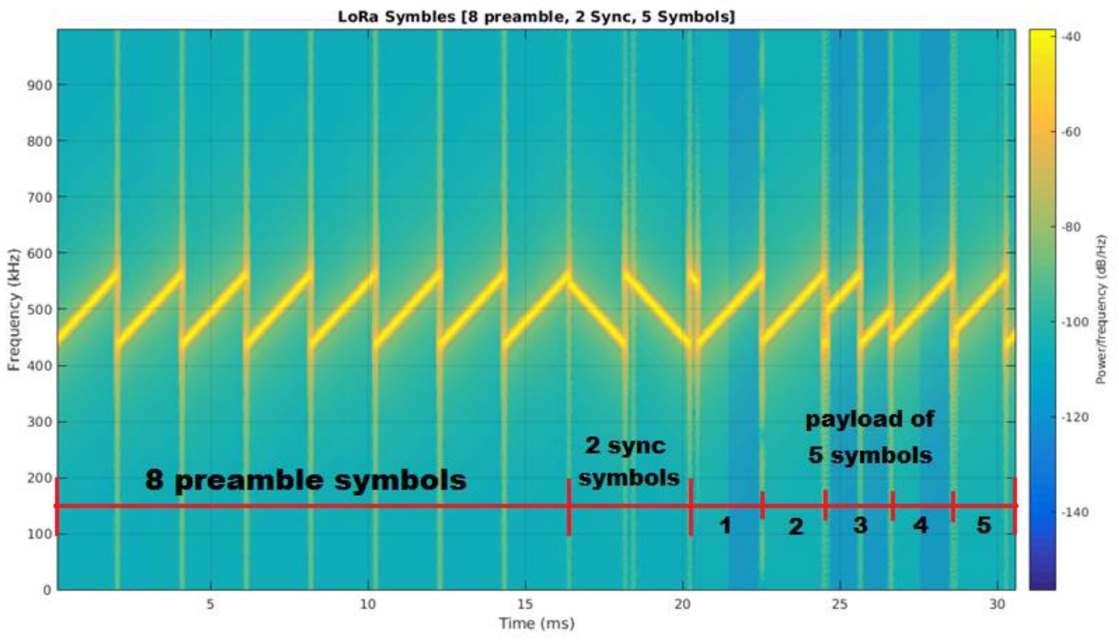


Figure 24 - An example of a LoRa message

From this figure we can observe that every symbol in the payload is not a simple up- or down-chirp signal. It looks as if they cut the up-chirp somewhere along its length and pasted the left section to the right. The closer this cut is to the left of the signal, the higher the value. As mentioned earlier, the symbol is divided in 2^{SF} chips, so if the data has 128 chips, this example payload could mean something like "115, 128, 64, 128, 21".

This technology has a very clear limitation: it can send very short messages and is not suitable for long strings of text. In order to deliver complex messages, we need to format them in a specific way to be sent and convert them again on the receiving end.

On the other hand, the advantages of LoRa is the low price of the devices, the low power consumption if used correctly, and the large range of its communications, as long as there is a direct line of sight to the antenna.

2.1.5. Power source

Every device needs its power source. Our microcontroller, the CubeCell, has a battery connector already incorporated in the device. However, every microcontroller has its way to connect it from a power source.

There is another issue: how much the device consumes. LoRa devices are famous for their low consumption rates, but we can obviate the question altogether by attaching a solar panel to the device. Conveniently, the CubeCell also has a way to use the solar panel to recharge the batteries, and it manages it on its own.

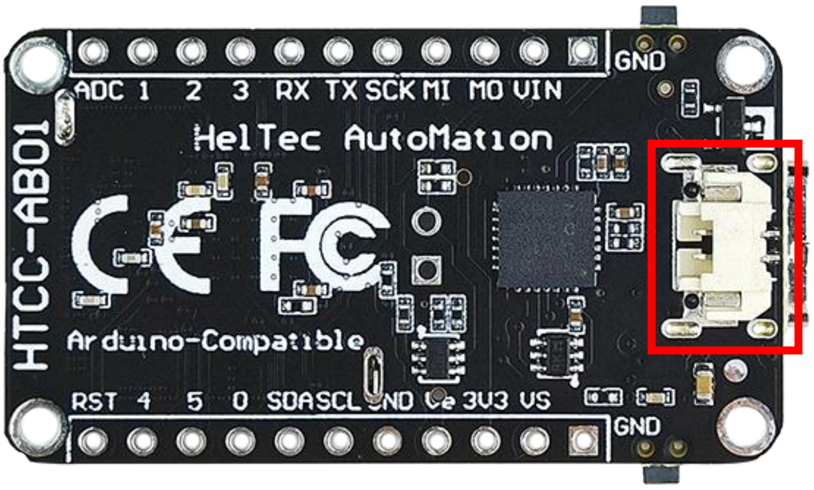

GND
ADC 1 2 3 RX TX SCK MI MO VIN
HelTec AutoMation
HTCC-AB01
Arduino-Compatible
RST 4 5 0 SDA SCL GND Ve 3V3 VS
GND

Figure 25 - Battery port of Heltec Cubecell HTCC-AB01

For other alternative microcontrollers, for instance, Adafruit makes small shields that can be attached to its Trinket devices to recharge the battery while an USB is connected.

2.2. Sending the data

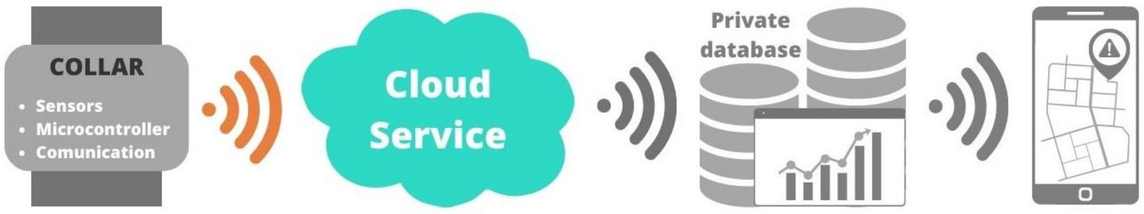


Figure 26 - Workflow: communication

Once our device has collected and processed the required data, we must use LoRa communication to send it somewhere else. In our case, our data will be temporarily stored in the "The Things Network" (TTN) servers, which will allow us to read the data from its website.

As mentioned before in the "Communications" section, we must prepare the message in a specific way in order to send it correctly. We must separate the payload in byte-sized pieces and instruct TTN how to reconstruct the message using a "payload formatter".

At this point, we will be able to visualize the data via TTN, but that information will not be stored on the website forever. We must retrieve the data from TTN and send it to our private database.

2.3. Storing the data

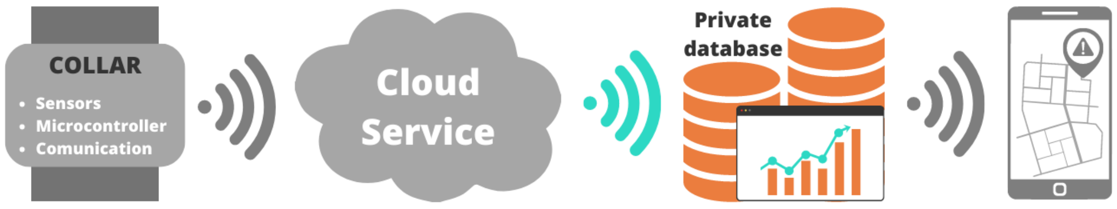


Figure 27 - Workflow: data storage

TTN allows for different "integrations" to connect its services to other platforms, such as MQTT, Webhooks, Azure IoT or LoRa Cloud. In our case, we will use the MQTT protocol to send the data to our own private database. We will make this connection using Balena Node-RED (20), a browser editor that will allow us to connect TTN and our database while filtering the message received.

2.4. Visualizing the data and interaction with the user

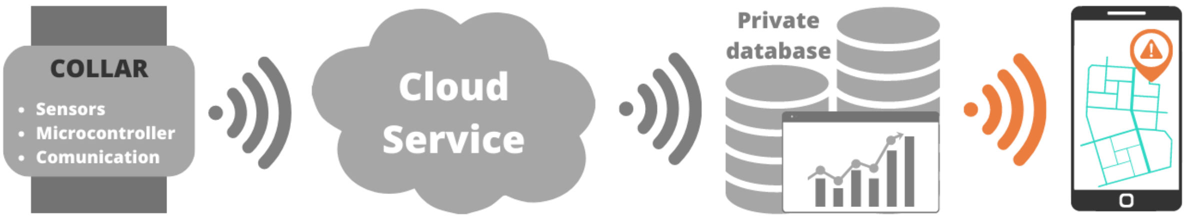


Figure 28 - Workflow: data visualization

At this point, we will have the desired variables (latitude, longitude, temperature, movement, etc.) stored in our database.

The last step would be making the data look appealing and easier to understand. For this purpose, we will use Grafana to make our own dashboard to visualize the gathered data.

As this is our own prototype designed to test a LoRa device for future use of the capabilities of this technology, our work will finish here. If a farmer decides to use this device in the future, we will have to make a small app so they could check this information from their phone.

Figure 29 - Example of a dashboard made with Grafana.

3. Detailed Design

3.1. Building the collar

3.1.1. Microcontroller

Arduino Pro Mini – not used in the end

Initially we did our tests using an Arduino Pro Mini and had every sensor working on its own. However, when using all the libraries together, we found out that the memory was not enough for such a large code.

The Arduino Pro Mini version we had selected was the one which runs at 3.3V and 8 Mhz, mainly due to its voltage level and compatibility with other devices. Therefore, the rest of the devices selected are compatible with these power levels.

One of the main advantages of using this device was its 3.3V regulated voltage pin, in which we can connect to many other devices directly without the need of using another voltage regulator or tension divider. Additionally, its small size and low price (about $12) was ideal for our project.

Its main limitation turned out to be its memory size of 32 KB, for which we needed to be careful when programming several devices attached to it. Additionally, it would be crucial to use specific libraries with reduced sizes. Its SRAM was only 2 KB and proved to be a limitation when testing every device together.

Heltec Cubecell HTCC-AB01

Due to the lack of memory of the Arduino Pro Mini, we switched to another microcontroller: Heltec Cubecell HTCC-AB01. We will refer to the device simply as "**Cubecell**" in the rest of the document.

In terms of implementing the rest of the devices we had to modify the code and use different libraries, but this development board is definitely a good change when compared to Arduino Pro Mini.

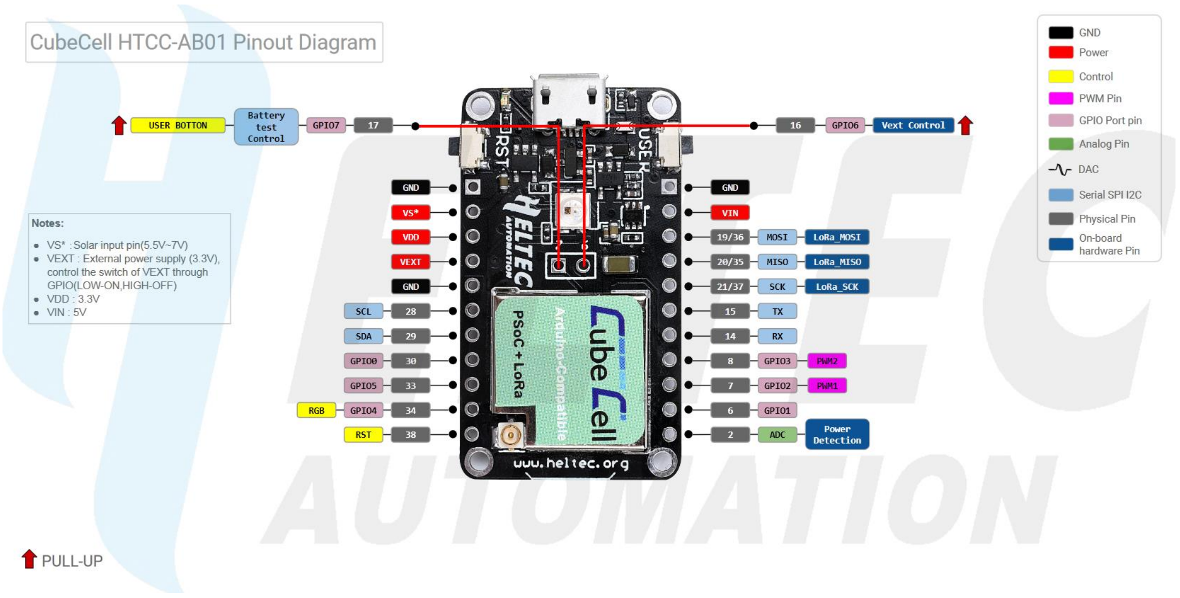


Figure 30 - Pinout Diagram for the Cubecell board

First off, the Cubecell development board already has LoRa capabilities integrated, so there is no need to add another device to the prototype. In contrast, with the Arduino Pro Mini we had used a RFM95 radio module, which used another 4 pins for communication via SPI (Nss, Sck, Miso, Mosi). With our Cubecell board, we do not need this device. This not only simplifies the prototype, but also frees the SPI port in case we want to add more sensors in the future.

Second, the Cubecell comes with a battery connector, and even has a solar panel input that can be used to recharge the battery. The board comes with the power management system already incorporated in the device.

Last, and most importantly, the available memory (128KB FLASH; 16KB SRAM) is more than enough to process our code. The two previous points solved small inconveniences, but this was a major issue. With the Arduino devices we had available we simply could not use such a large code.

The following are some of the most relevant aspects of Cubecell's Datasheet for our project. Additionally, it is worthy of note that there is a library that makes the device fully Arduino-compatible.

Parameters	Description
Master Chip	ASR6501 (48 MHz ARM® Cortex® M0+ MCU)
Frequency	863~923 MHz (We need 868 MHz for LoRa)
Low Power	Deep Sleep 3.5µA
Hardware Resource	UART x 1 (used for GPS); I2C x 1 (used for accelerometer); etc.
Memory	128KB internal FLASH; 16KB internal SRAM
Battery	3.7V Lithium(SH1.25 x 2 socket)
Dimensions	41.5 x 25 x 7.6 mm

3.1.2. Location – GPS module

The selected module for our location device is the **Neo-6M** GPS module. This device can be powered between 2.7 V and 3.6 V, making it fully compatible with our microcontroller device.

Communication with this module is done via UART (Universal Asynchronous Receiver-Transmitter), requiring just power and two pins connected to the microcontroller to work.

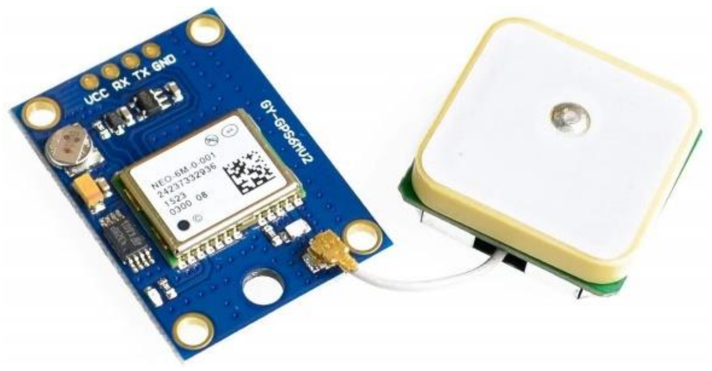


Figure 31 - The GPS module Neo-6M

The module has a resistant ceramic antenna, whose bottom side (the one without the "dot") is connected to the Ground pin. This will be relevant when creating a PCB.

3.1.3. Accelerometer, gyroscope, and temperature module

The selected module is the **MPU6050**, an Inertial Measurement Unit (IMU) with 6 degrees of freedom, combining a 3-axis accelerometer and a 3-axis gyroscope. It is a small and cheap device (around 6€) and is often used for diverse projects such as navigation, stabilization devices or angle measurements.

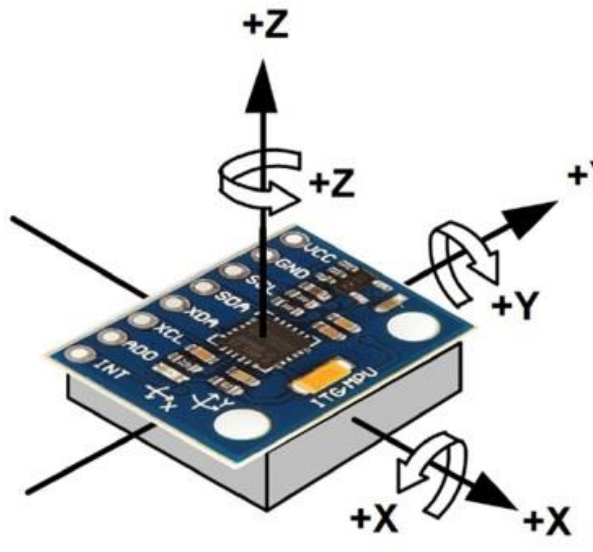


Figure 32 - Axes of the MPU6050 module.

The module can communicate with the microcontroller via I2C and can be connected directly to Arduino-compatible devices due to the pull-up resistance in the SCL and SDA pins. Thanks to an internal 3.3V voltage regulator, it could be connected to a higher voltage if needed, but its regular input voltage of 2.375 V - 3.46 V is already compatible with our microcontroller.

This device not only can output the 6 DoF measurements, but also has an integrated temperature sensor with a digital output, which makes it very convenient for the device and adds additional features without increasing its complexity with additional parts.

3.1.4. Communication

The CubeCell development board already has its own integrated LoRa communications system, so we do not need to add any additional components here.

However, it is worth mentioning that during part of the development of this project we have used an Arduino Pro Mini and a RFM95 module, which is able to handle LoRa transmissions. In order to communicate with Arduino, the module used the SPI bus. Additionally, it could be connected directly to 3.3 V Arduino Pro controllers.

If this project is replicated in the future using a microcontroller lacking LoRa modules, RFM95 is a good way to handle these transmissions.

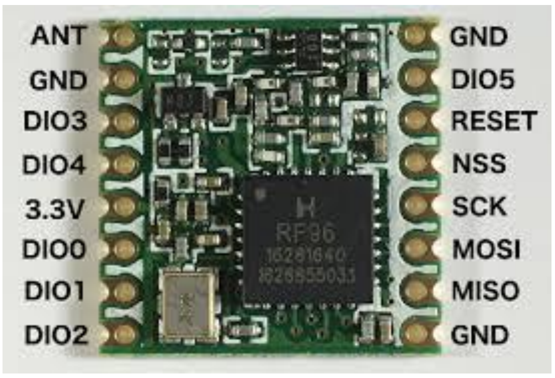


Figure 33 - The LoRa module RFM95 (not used in the end)

3.1.5. Power source

The CubeCell development board has a battery input port, to which we can directly connect a 3.7 V LiPo battery. Not only does the board have a dedicated battery input, but it also has another interesting feature: if you connect a solar panel using its VS and GND pins, the solar panel will recharge the battery due to an integrated power management system.

Figure 34 - Lipo battery

We have found a small 5 V solar panel that fits easily in our board and added a two-wire connector to the protoboard and PCB. With this, we need not worry about the battery's lifespan, as it will be recharged daily by the solar panel.

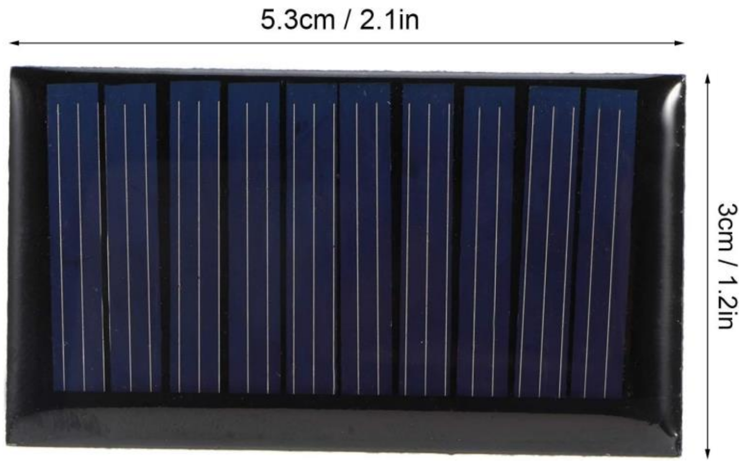


Figure 35 - Solar panel

3.1.6. PCB

While all the tests during development were made on a protoboard, we decided to make a final product by creating a custom PCB. One of the layers will have the devices that must be in view of the sky (the solar panel and the antenna of the GPS module), while the other layer has the rest of the devices. The accelerometer must be placed here as well, so its temperature sensor is close to the animal's skin.

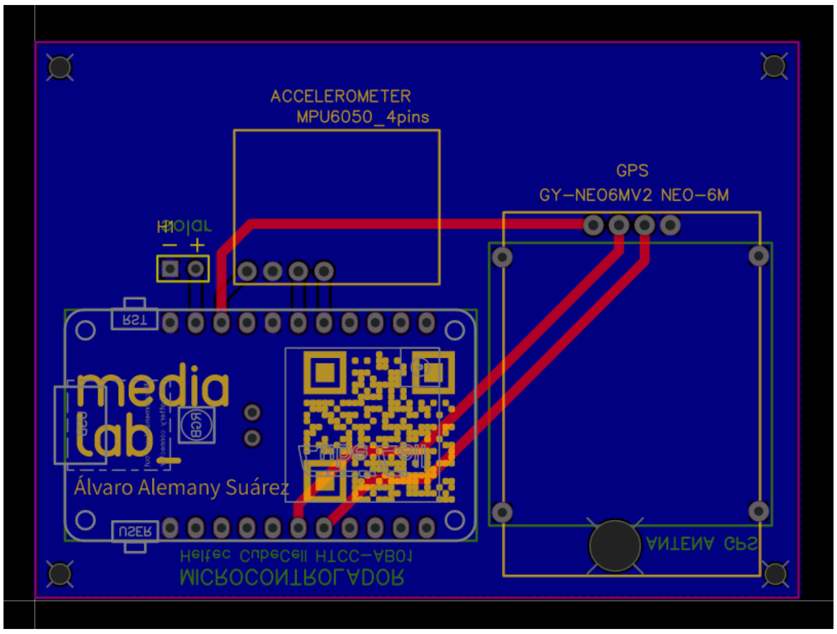


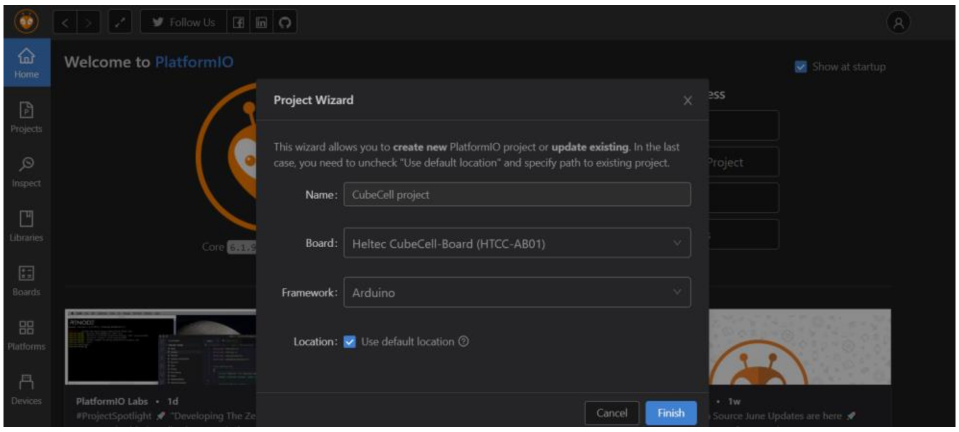
Figure 36- PCB created for the prototype.

3.2. Programming the device

As mentioned before, the CubeCell microcontroller is fully compatible with Arduino libraries and programming can be easily done with the standard Arduino IDE. However, due to the large number of libraries to manage, we have used **Visual Studio Code** and its extension **PlatformIO** to develop our code.

The reason for this is the fact that PlatformIO is a project-based system, in which we can customize every single project and add different libraries to each version. It is easy to manage the installed boards and libraries, of which we can find a great variety.

After installing VSC and its extension PlatformIO, we create our project. We selected "Heltec CubeCell-Board (HTCC-AB01)" as our board and "Arduino" as our framework.



Project Wizard

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name: CubeCell project

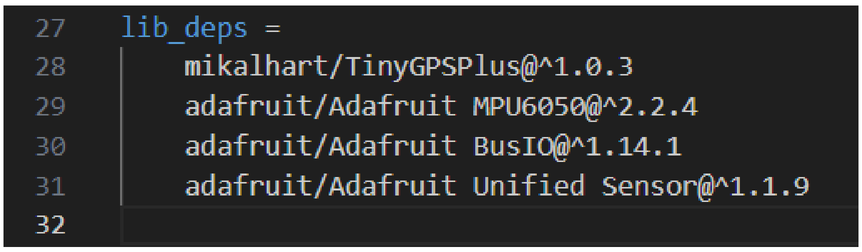
Board: Heltec CubeCell-Board (HTCC-AB01)

Framework: Arduino

Location: ☑ Use default location

Figure 37 - Screenshot of PlatformIO, creating a new project.

We are ready to start developing our code, but first we must install our libraries. For this, we go to Platformio → Libraries → We search for the name of the library → Add to project. Then we select the project we are working on and give it some time to install the library. Additionally, if you go to the file "platformio.ini", you will observe that the installed library appears at the bottom of the file.



```
lib_deps =
    mikalhart/TinyGPSPlus@^1.0.3
    adafruit/Adafruit MPU6050@^2.2.4
    adafruit/Adafruit BusIO@^1.14.1
    adafruit/Adafruit Unified Sensor@^1.1.9

```

Figure 38 - library dependencies

3.2.1. Accelerometer

The module will use the **Adafruit MPU6050** library. Additionally, this library depends on two other libraries, which will be installed automatically and included within the first one: **Adafruit BusIO** and **Adafruit Unified Sensor**.

```
#include <Adafruit_MPU6050.h> // for the accelerometer
```

The MPU6050 library has many useful functions already defined. For the sake of simplicity and having a more modular code, we will define some new functions that will be called inside the "setup" or "loop" function afterwards.

```
Adafruit_MPU6050 mpu;           // We create an object for the sensor.
bool mov = 0;                  // global variable for the presence of movement.
void Init_Motion();            // initiates accelerometer as a motion sensor.
void Detect_Motion();          // detects motion and prints the data on screen.
```

The first function we create is `Init_Motion()`, which starts by trying to initialize the sensor. If it is unable to do so, shows a message on screen and gets stuck on purpose, forcing a restart of the system to work again.

Otherwise, the function goes on, establishing the accelerometer as a motion detector. This will generate an interruption when a movement is detected based on the established threshold and duration.

```
void Init_Motion() // It initializes the MPU6050 as a motion sensor.
{
  if (!mpu.begin()) {           // Tries to initialize the sensor. If it can't:
    Serial.println("Failed to find MPU6050 chip. Check wiring and restart");
    while (1)       delay(10);  // gets stuck here forever, requiring a restart.
  }
  Serial.println("MPU6050 Found!");  // If the begin() call was successful, we continue.

  // Now we configure the motion detection
  mpu.setHighPassFilter(MPU6050_HIGHPASS_0_63_HZ); //5, 2.5, 1.25, 0.63, Unused or Hold.
  mpu.setMotionDetectionThreshold(1);
  mpu.setMotionDetectionDuration(20);
  mpu.setMotionInterrupt(true);                    // enables motion interrupt.
}
```

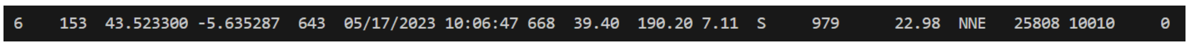
The second function is called `Detect_Motion()` and is used to print all the information on screen. This will help when "debugging", but in the end we will just use the first lines in our functions, without printing anything.

```
void Detect_Motion()
{
  if(mpu.getMotionInterruptStatus())
  {
    /* Get new sensor events with the readings */
    sensors_event_t a, g, temp;
    mpu.getEvent(&a, &g, &temp);

    /* Print out the values */
    Serial.print("AccelX:");     Serial.print(a.acceleration.x);   Serial.print(", ");
    Serial.print("AccelY:");     Serial.print(a.acceleration.y);   Serial.print(", ");
    Serial.print("AccelZ: ");    Serial.print(a.acceleration.z);   Serial.print(",   ");
    Serial.print("GyroX: ");     Serial.print(g.gyro.x);           Serial.print(", ");
    Serial.print("GyroY: ");     Serial.print(g.gyro.y);           Serial.print(", ");
    Serial.print("GyroZ: ");     Serial.print(g.gyro.z);           Serial.println();
  }
  delay(10);
}
```

3.2.2. GPS

In order to obtain our location data, we will use a library called **TinyGPS++** by Mikal Hart. While the original TinyGPS library is functional, parsing the data in the TinyGPS++ is easier and has more features. Additionally, as memory is not a limiting factor anymore, working with this library is much more convenient for extracting the appropiate data.



```
6    153  43.523300 -5.635287  643  05/17/2023 10:06:47 668  39.40  190.20 7.11  S     979      22.98  NNE   25808 10010     0
```

Figure 39 - Example of GPS data decoded in the serial monitor.

We also need another library called "SoftSerial" (which replaces Arduino's "Software Serial") to send data via UART. These are the lines included at the beginning of the code:

```
#include <SoftSerial.h>                             // for the GPS module
#include <TinyGPS++.h>                              // for the GPS module
```

We proceed to declare the variables that the GPS will use, the functions that will be called in the future, and the pins of the CubeCell that will be used for communication.

```
// GPS MODULE -----------------------------------------------------------------------
TinyGPSPlus gps;                  // object for the GPS
long   lat = 0 , lon = 0 ;    // global variables for latitude and longitude
int    age = 0;       // "age" of the signal : time elapsed since the last GPS fix
void   Smart_Delay(unsigned long ms);//works like delay but still encodes GPS data.
void   Get_GPS_Data();      // function that will read, process and store GPS data.
void   Wait_for_Data(unsigned long ms); // returns when valid data is received.
static const uint32_t GPSBaud = 9600;       // baud rate for communicating with the GPS.
softSerial ss(GPIO3 ,    /* Pin of the CubeCell used as TX -> brown wire -> to GPS-RX */
              GPIO2 );   /* Pin of the CubeCell used as RX -> green wire -> to GPS-TX */
```

One of the main issues we encounter when programming a GPS module is that it must be constantly listening to data. If it misses a message, then it will not be able to perform a correct trilateration of the coordinates and time. This is where the first function, `Smart_Delay()`, is used. This custom version of `delay()` ensures that the GPS object is being constantly "fed". It records the current time, in milliseconds, and whenever it receives a NMEA sentence, it encodes and stores the data. The rest of the time, it does nothing, just like "delay".

```
void Smart_Delay(unsigned long ms)  // ensures that the gps object is being "fed".
{
  unsigned long start = millis();// records the current time in milliseconds
  do {                           // and does the two following lines
    while (ss.available())
      gps.encode(ss.read());
  } while (millis() - start < ms);  // until the specified number of ms has passed
}
```

A similar function is `Wait_for_Data(ms)`, which works almost identically to the previous one. However, at the moment that it detects valid data with an "age" lower than 1 second (which is an indication of a well decoded and stable flow of information from the satellites), then it returns. You could think of this function as "try to gather data for a certain amount of time. If you manage to get the data, return immediately. If you don't get any good data, then return regardless."

```
static void Wait_for_Data(unsigned long ms)
{
  uint32_t start = millis();    // records the current time in milliseconds.
  while( (millis()-start) < ms )  // As long as the time hasn't elapsed
  {                                 // it does the following lines:
    while (ss.available() > 0) gps.encode(ss.read());  // encodes new data
    if( gps.location.age() < 1000 )         // and if the data is valid
    { Serial.println("Data acquired"); // prints the message,
      break;           // and gets out of the function.
    }
  }
}
```

The next function is `Get_GPS_Data()`, which extracts the necessary variables from the GPS object created earlier, showing the **HDOP** (Horizontal Dilution Of Precision, the larger the number, the worse the precision of the data), **latitude**, **longitude** and **age** of the measurement. **Age** is quite an important variable, as it tells us how long it has been since the last valid sentence has been decoded. If it becomes too large, it can signify the loss of connection. We have seen it in the previous function as a sort of validity check.

In the function we can observe the conversion of data as well, from a **float** to a **long** integer. This is done in order to easily divide it into bytes when the message is prepared. In order not to lose the 6 decimal digits of precision by truncating the float, first we multiply it by a million.

```
void Get_GPS_Data()
{
    Serial.print("Prec: ");  Serial.print(gps.hdop.hdop());

    float flat = gps.location.lat()*1000000;
    lat = static_cast<long>(flat);
    Serial.print("  Lat_int: ");  Serial.printf("%d",lat);

    float flon = gps.location.lng()*1000000;
    lon = static_cast<long>(flon);
    Serial.print("  Lon_int: ");  Serial.printf("%d",lon);

    age = gps.location.age();
    Serial.print("  Age: ");      Serial.printf("%d",age);

    Serial.println();
}
```

3.2.3. Sending the data to LoRaWAN

There are two main functions that will handle LoRaWAN communication. The first one, LoRaWAN_Loop(), is exactly what is needed to be in the regular loop() function from Arduino codes. However, I believe having it in a separate function and calling it in the loop makes it look cleaner and is useful for debugging when we are not interested in sending the data to LoRaWAN yet.

```
// LoRaWAN ---------------------------------------------------------------------------
void  LoRaWAN_Loop(); // handles the states of the device, and should be alone in loop()
void  Prepare_Data( uint8_t ); //the most important function, prepares the data to send
```

First, we are going to talk about LoRaWAN_Loop() . We could literally copy the inside of this function inside loop(), and the prototype should work the same way. This is basically a state manager for the device, with different orders when it needs to initialize, join TTN, send the data, schedule the next transmission or turn LoRaWAN communication off.

```
void LoRaWAN_Loop()
{
  switch( deviceState )                      // looks at the state of the CubeCell device.
  {
    case DEVICE_STATE_INIT: // at the beginning, it will be in its initialization state.
    {
      printDevParam();                            // We show its parameters on screen.
      LoRaWAN.init(loraWanClass,loraWanRegion); // And initialize the device.
      deviceState = DEVICE_STATE_JOIN;            // Then switches to the next state.
      break;
    }

    case DEVICE_STATE_JOIN:             // in this state, it tries to connect to TTN.
    {
      LoRaWAN.join();            // trying to establish the connection again and again.
      break;// Only after a successful connection will the device get out of this state.
    }

    case DEVICE_STATE_SEND: // When TTN is ready to receive the message...
    {
      Prepare_Data( appPort );          // We call the function to prepare the data.
      LoRaWAN.send();                             // and send the payload.
      deviceState = DEVICE_STATE_CYCLE;//when the message is sent, go to the next state.
      break;
    }

    case DEVICE_STATE_CYCLE:// we define when we will send the next message.
    {
      txDutyCycleTime = appTxDutyCycle + randr( 0, APP_TX_DUTYCYCLE_RND );
       // The time has a minimum part and a random addition
      LoRaWAN.cycle(txDutyCycleTime); // then, we "set an alarm" to wake up LoRaWAN.
      deviceState = DEVICE_STATE_SLEEP;         // and go to sleep
      break;
    }

    case DEVICE_STATE_SLEEP:// all we do is sleep until txDutyCycleTime has passed.
    {
      LoRaWAN.sleep();
      break;
    }

    default:  // if no case is selected or it is not one of those defined before
    {
      deviceState = DEVICE_STATE_INIT;          // we restart the process
      break;
    }
  }
}
```

And for the last function in the code, we will talk about the most important one. This function will wake up the sensors, read the data, store them, chop them into pieces and prepare the message to be sent to TTN. If the previous functions were not defined, this function would be hundreds of lines long.

```
void Prepare_Data( uint8_t port )
{
  digitalWrite(Vext, V_ON);                    // we ensure the sensors are on
  Smart_Delay(10);             // and give them a small time to reach steady state.
  ss.begin(9600);         // establishes the SoftSerial connection with the GPS sensor.
  Serial.println("GPS Searching..."); // and prints the message on screen.

  Wait_for_Data(10000);    // we give it some time to get a proper fix.

  if(gps.location.age() < 1000)  // if we get a proper fix of the GPS
  { Get_GPS_Data(); }            // we store the desired data and print it on screen.
  else                           // if we can't get a proper fix of the GPS data…
  {                                   // we do two things:
    Serial.println("No GPS signal");   // - we print a warning on screen.
    Smart_Delay(2000);                 // - and wait                         .
  }

  if(mpu.getMotionInterruptStatus()){ // we obtain the data from the accelerometer.
    Serial.println("Motion: Yes");       // if motion is detected, we print this…
    mov = 1;                              // and update the variable
  }
  else{                                // if motion was not detected by the filter…
    Serial.println("Motion: No");         // we print this message…
    mov = 0;                              // and update the variable
  }

  sensors_event_t a, g, temp;      // create a structure for a single sensor event.
  mpu.getEvent(&a, &g, &temp);     // and read the data.

  Serial.print("Temp: ");         // outputs the temperature
  int temp_cents_deg = static_cast<int>(temp.temperature*100);  // and transforms it
  Serial.print(temp_cents_deg); Serial.println(" cents of a ºC");

  // Now we are going to divide our data into bytes. There is a payload formatter in TTN
that will reconstruct the message.

  appDataSize = 11;           // Number of characters that are going to be sent

  appData[3] = lat;           // as the latitude is going to be sent as a "long" integer
  appData[2] = lat >> 8;      // it has 32 bits of length, that is, 4 bytes
  appData[1] = lat >> 16;     // so we separate it into four byte pieces
  appData[0] = lat >> 24;     // in order to reconstruct it later in TTN

  appData[7] = lon;           // we do the same for the longitude
  appData[6] = lon >> 8;
  appData[5] = lon >> 16;
  appData[4] = lon >> 24;

  appData[8] = highByte(temp_cents_deg); // we do the same for the temperature, 2 bytes
  appData[9] = lowByte(temp_cents_deg);  // which means the maximum temp is 655.35 ºC

  appData[10] = mov;     // this will be used as a boolean to indicate movement.
}
```

The reason why we must separate the data in this way, divided in byte-sized pieces, is the way that LoRaWAN transmits the message. In a later section we will see how a payload formatter reconstructs the data back into its original form in TTN. The method using to divide is bitwise shifting. For instance, the latitude 43.523933 would be transformed into an integer by multiplying it by 1,000,000, resulting 43523933 . This number will have a binary representation (adding zeros to the left to complete a 32-bit integer).

Binary representation above, and equivalent Hexadecimal number below:

0000	0010	1001	1000	0001	1111	0101	1101
0	2	9	8	1	F	5	D

Thus, this value is separated into 4 bit-sized numbers: 02, 98, 1F, 5D , and sent as-is to TTN, where it will be reconstructed afterwards.

3.2.4. Setup and Loop

Arduino devices are usually programmed with two functions: setup and loop. The names are quite intuitive: **setup** is a function that is run through once at the beginning and configures the controller and connected devices, and **loop** runs in a continual loop, as the name indicates.

In the setup function, we establish the necessary configurations for the devices.

```
void setup()
{
  pinMode(Vext,OUTPUT);     // We will decide when to turn the external voltage ON or OFF
  digitalWrite(Vext, V_ON); // now we turn it ON.
  Serial.begin(9600); // We begin serial communication with the screen, at 9600 baud.
  Wire.begin();       // We start the Wire communication with the accelerometer.
  Init_Motion();      // And configure it with the settings defined in this function-
  ss.begin(GPSBaud); // We start communication with the GPS module using SoftSerial.
  deviceState = DEVICE_STATE_INIT;  // We establish the initial state of the Cubecell
  LoRaWAN.ifskipjoin();  // And use the stored OTAA credentials to join session.
}
```

And in the loop function we simply call the LoRaWAN_Loop() function. As mentioned before, it is done this way for ease of debugging and testing different functions while TTN is not available or not needed. As a reminder, this function will check the state of the device, and when the time is right, it will have to prepare the message, calling the functions for the other sensors.

```
void loop()
{
  LoRaWAN_Loop();       // we call the LoraWAN loop.
}
```

3.2.5. Separate TTN configuration

There is one last detail to discuss here regarding programming. In order for the CubeCell to join TTN, it must have the right values for identifying the device and its “keys” to establish the session. This information can be included in the main.cpp piece of code without any issue, but it is good practice to store them in a separate file. We called this file “TTNvalues.h”, and can be easily integrated in the code by adding the following line:

```
#include "TTNvalues.h"                                  // LoRa
```

Why do we do this if the result is the same? Because anybody with the same keys we use in this project could copy them into their device and use them to input their data into our database, whether accidentally or with malicious intent. We will show the structure of the file below but consider that all the values have been replaced by zeroes in order not to reveal critical information.

```
// OTAA parameters, all in "big endian" (msb)
uint8_t appEui[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
uint8_t devEui[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
uint8_t appKey[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
// ABP parameters, all in "big endian" (msb)
uint8_t nwkSKey[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00};
uint8_t appSKey[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00};
uint32_t devAddr =  ( uint32_t )0x000000000;

uint16_t userChannelsMask[6]={ 0x00FF,0x0000,0x0000,0x0000,0x0000,0x0000 }; //LoraWan
channelsmask, default channels 0-7*/

LoRaMacRegion_t loraWanRegion = ACTIVE_REGION;// defined in platformio.ini .
DeviceClass_t  loraWanClass = LORAWAN_CLASS;  // LoraWan Class.
uint32_t appTxDutyCycle = 15000;              // transmission duty cycle in [ms].
bool overTheAirActivation = LORAWAN_NETMODE; // OTAA or ABP, defined in platformio.ini .
bool loraWanAdr = LORAWAN_ADR;               // ADR enable
bool keepNet = LORAWAN_NET_RESERVE;          // set LORAWAN_Net_Reserve ON, the node
could save the network info to flash, when node reset not need to join again
bool isTxConfirmed = LORAWAN_UPLINKMODE; // Indicates if the node is sending confirmed
or unconfirmed messages
uint8_t appPort = 1;                        // Application port
uint8_t confirmedNbTrials = 4;
```

3.3. Gathering the data

3.3.1. The Things Network

The Things Network (**TTN**) is "a global collaborative Internet of Things ecosystem that creates networks, devices and solutions using LoRaWAN"(13). They have an open and decentralized network in which to use LoRaWAN applications and devices. Using TTN is free and has a wide variety of learning activities and tutorials to get started with LoRaWAN.

Creating an account is free. In our case, Medialab already has a TTN account, and we will integrate our system within it. In the case that anyone else wanted to replicate this project, they would have to do so with their own account.

After entering TTN with the right credentials, we have to access the **Console** and select the region we want to work on, in this case "**Europe1**". Then, we select "go to **applications**" and select the name of the application we want to add or create a new one.

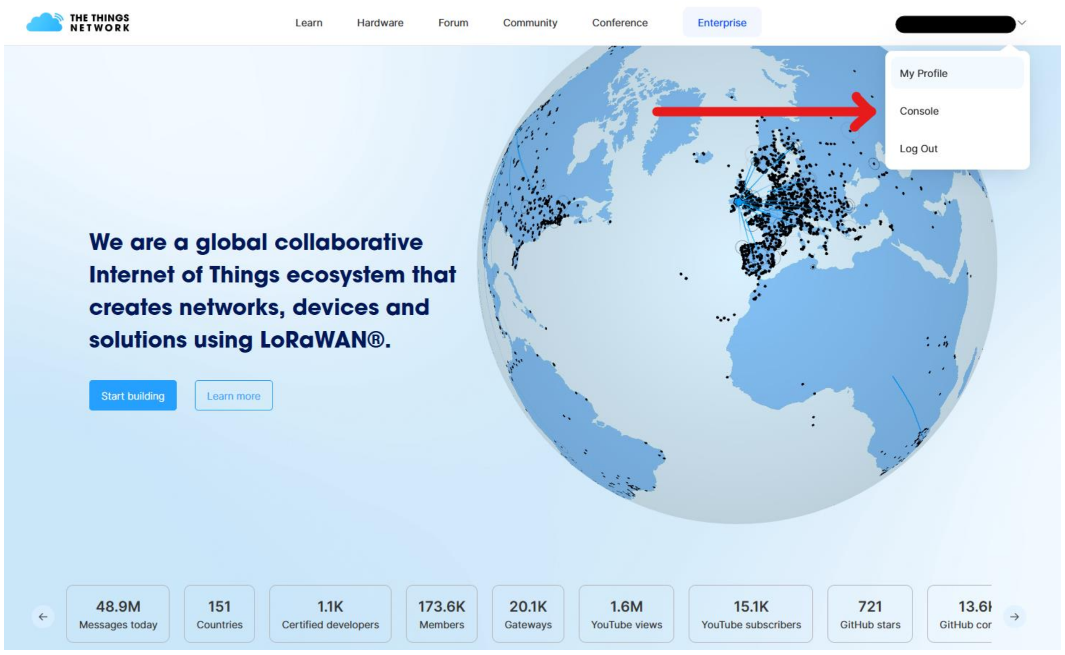


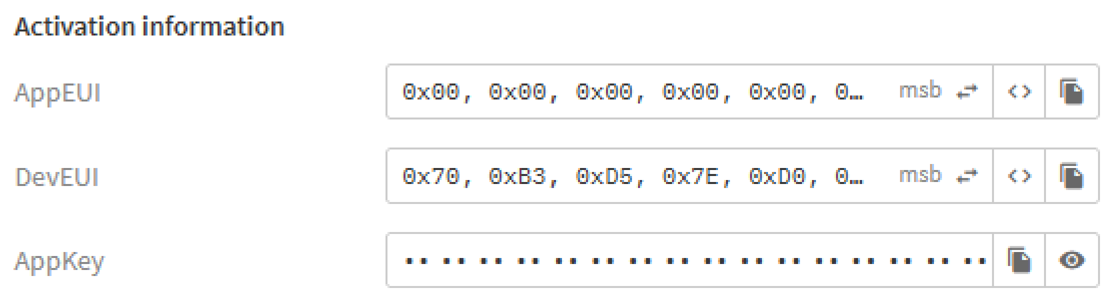
Figure 40 - TTN webpage once you have logged in.

Once inside the application, we can create and modify the end devices. For instance, we can configure the application to work in a certain way for ten different location devices. If you are using Arduino + RFM95, they are not in the list of available devices, so you will need to add your end device manually. For our case with CubeCell, we simply need to select from some options in a list. We go to “Register end device” and select the following options:

Input method	Select the end device in the LoRaWAN Device Repository
End device brand	HelTec AutoMation
Model	HTCC-AB01 (Class A OTAA)
Hardware Ver.	Unknown ver.
Firmware Ver.	1.0
Profile (Region)	EU_863_870
Frequency plan	Europe 863-870 MHz (SF9 for RX2 – recommended)

Now we need to enter the **JoinEUI** of the device. We can start by simply filling it with zeroes. Then, we will generate the **DevEUI** and **AppKey** for the device. We can additionally name our end device with an “End device ID”, which has its DevEUI included in the name as a default option but does not need to appear in the final name.

Once finished with the configuration, we will be able to see our device in the list. If we click on it, in the “Overview” tab, we will see our activation information. These numbers (**AppEUI**, **DevEUI**, **AppKey**) will be used in the code of our device, as if they were the keys or passwords required for communication with TTN. This information is stored in the file called “TTNvalues.h” . Make sure the format is **msb** (most significant bit):


Activation information

AppEUI	0x00, 0x00, 0x00, 0x00, 0x00, 0… msb
DevEUI	0x70, 0xB3, 0xD5, 0x7E, 0xD0, 0… msb
AppKey	•• •• •• •• •• •• •• •• •• •• •• •• •• •• •• ••

Figure 41 - Extract from TTN where Activation Information is stored.

After configuring the code in Visual Studio Code and preparing the message to be transmitted, we need to return to TTN and configure the "**payload formatter**". This is a piece of code that tells TTN how to interpret the received message.

```
function decodeUplink(input) {
  var bytes = input.bytes;
  var lat_millon = bytes[3] | bytes[2] << 8 | bytes[1] << 16| bytes[0] << 24;
  var lon_millon = bytes[7] | bytes[6] << 8 | bytes[5] << 16| bytes[4] << 24;
  var temp_cents = bytes[9] | bytes[8] << 8 ;
  var lat = lat_millon / 1000000;
  var lon = lon_millon / 1000000;
  var temp = temp_cents / 100;
  var mov = bytes[10];
  var id = 1;   // este número identifica a este dispositivo concreto

  if (lat === 0)  lat = "invalid";
  if (lon === 0)  lon = "invalid";

  return
  {
       data: { id,  lat, lon, temp, mov },: [],
       warnings: [],
       errors: []
  };
}
```

A way of explaining it in human language could be the following:

" You are going to receive a series of data composed of one byte each. The first four bytes will show the latitude of the device, in millionths of a degree. You must shift the data the following way to reconstruct the whole number.

Take the 1st byte and shift it 24 bits (3 bytes) to the left.	***02** 00 00 00*
Take the 2nd byte and shift it 16 bits (2 bytes) to the left.	*00 **98** 00 00*
Take the 3rd byte and shift it 8 bits (1 byte) to the left.	*00 00 **1F** 00*
Take the 4th byte as is.	*00 00 00 **5D***
Now, add these numbers together.	*02 98 1F 5D*
And assign this value to the variable lat_millon	*43523933*
Later, get the actual variable by dividing it by a million	*43.523933 "*

We repeat a similar process with other variables, until we have the whole message reconstructed. Then, we tell the payload formatter to return the message in the desired order.

Once the payload formatter is dealt with in TTN, we need to establish a way to deliver those data into an external database. TTN works perfectly to read live data, but information in its servers is volatile. We will connect the stream of information TTN provides with Node-RED, a program that will act as the link and "translator" between TTN and our database.

3.3.2. Node-RED configuration

Node-RED is, in essence, a programming tool that has a browser-based editor and uses function blocks called nodes that connect to each other and will be used to manipulate the long stream of characters received from TTN.

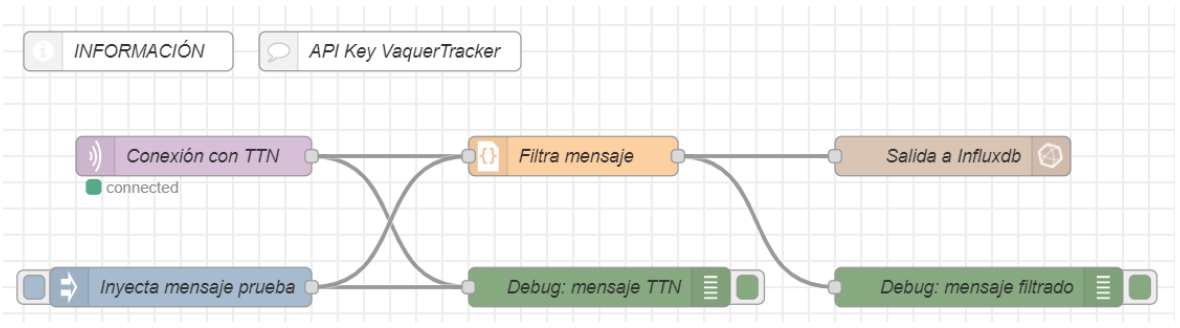


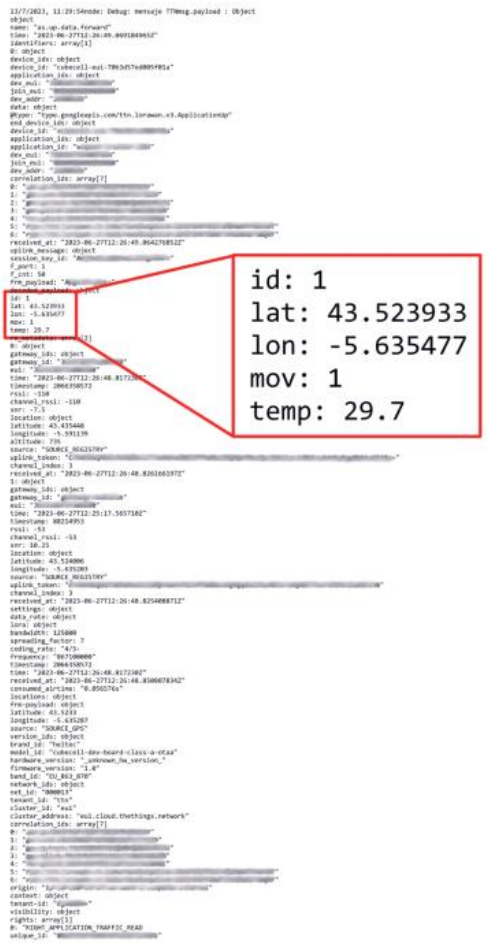
Figure 42 - An example of Node-RED blocks

Node-RED is integrated in a Raspberry device in Medialab, and thus can only be accessed and configured locally, by connecting to its Wi-Fi network and entering a specific address in the browser.

The information will be extracted from the TTN server using MQTT, a widely used IOT communication protocol. The only thing we need to know is how to configure it using TTN, which is a simple block with a couple of configuration options. We add the server in which the MQTT broker is installed, and we are ready to receive the long stream of data.

The subsequent functions gradually filter out and simplify the messages received. For instance, we already know the address of the device, the AppEUI, DevEUI, and metadata such as the fact that we are using LoRa modulation, our data rate has spreading factor 7, etc.

What we are interested in is isolating the required variables such as latitude and longitude and discarding the rest. Once we have our selected data, we must format it in a way that we can send it to the web.



```
13/7/2023, 11:29:54node: Debug: mensaje TTNmsg.payload : Object
object
name: "as.up.data.forward"
time: "2023-06-27T12:26:49.069184965Z"
identifiers: array[1]
0: object
device_ids: object
device_id: "cubecell-eui-70b3d57ed005f01a"
application_ids: object
dev_eui: "[redacted]"
join_eui: "[redacted]"
dev_addr: "[redacted]"
data: object
@type: "type.googleapis.com/ttn.lorawan.v3.ApplicationUp"
end_device_ids: object
device_id: "[redacted]"
application_ids: object
application_id: "[redacted]"
dev_eui: "[redacted]"
join_eui: "[redacted]"
dev_addr: "[redacted]"
correlation_ids: array[7]
0: "[redacted]"
1: "[redacted]"
2: "[redacted]"
3: "[redacted]"
4: "[redacted]"
5: "[redacted]"
6: "[redacted]"
received_at: "2023-06-27T12:26:49.064276852Z"
uplink_message: object
session_key_id: "[redacted]"
f_port: 1
f_cnt: 58
frm_payload: "[redacted]"
decoded_payload: object
id: 1
lat: 43.523933
lon: -5.635477
mov: 1
temp: 29.7
rx_metadata: array[2]
0: object
gateway_ids: object
gateway_id: "[redacted]"
eui: "[redacted]"
time: "2023-06-27T12:26:48.817230"
timestamp: 2060358572
rssi: -110
channel_rssi: -110
snr: -7.5
location: object
latitude: 43.435448
longitude: -5.591139
altitude: 735
source: "SOURCE_REGISTRY"
uplink_token: "[redacted]"
channel_index: 3
received_at: "2023-06-27T12:26:48.826166197Z"
1: object
gateway_ids: object
gateway_id: "[redacted]"
eui: "[redacted]"
time: "2023-06-27T12:25:17.5657182"
timestamp: 88214953
rssi: -53
channel_rssi: -53
snr: 10.25
location: object
latitude: 43.524006
longitude: -5.635203
source: "SOURCE_REGISTRY"
uplink_token: "[redacted]"
channel_index: 3
received_at: "2023-06-27T12:26:48.825408871Z"
settings: object
data_rate: object
lora: object
bandwidth: 125000
spreading_factor: 7
coding_rate: "4/5"
frequency: "867100000"
timestamp: 2060358572
time: "2023-06-27T12:26:48.8172302"
received_at: "2023-06-27T12:26:48.850607834Z"
consumed_airtime: "0.056576s"
locations: object
frm-payload: object
latitude: 43.5233
longitude: -5.635207
source: "SOURCE_GPS"
version_ids: object
brand_id: "heltec"
model_id: "cubecell-dev-board-class-a-otaa"
hardware_version: "_unknown_hw_version_"
firmware_version: "1.0"
band_id: "EU_863_870"
network_ids: object
net_id: "000013"
tenant_id: "ttn"
cluster_id: "eu1"
cluster_address: "eu1.cloud.thethings.network"
correlation_ids: array[7]
0: "[redacted]"
1: "[redacted]"
2: "[redacted]"
3: "[redacted]"
4: "[redacted]"
5: "[redacted]"
6: "[redacted]"
origin: "[redacted]"
context: object
tenant-id: "[redacted]"
visibility: object
rights: array[1]
0: "RIGHT_APPLICATION_TRAFFIC_READ"
unique_id: "[redacted]"
```

```
id: 1
lat: 43.523933
lon: -5.635477
mov: 1
temp: 29.7
```

Figure 43 - Length of the TTN message vs the actual information needed

3.4.Data storage in a database

We will use the HTTP request with the GET method. For those foreign to informatics, this may sound confusing, but it actually is something we indirectly use every day. In the end, Node-RED will get the latitude and longitude values and construct a link very similar to those we write in a browser. For instance:

www.medialab-uniovi.es/post_vaquer_tracker.php?id='1'&lat='43.523933'&lon='-5.635477'&t='29.7'&m='1'

As a recap, what we have received from TTN is a massive text that contained a vast amount of information, in which we were only interested in the values for latitude and longitude of the device. The functions in Node-Red filter these data and arrange it in a single string of text, that roughly means "go to this server (www.medialab-uniovi.es), look for this specific file (post_vaquer_tracker.php) and tell them the values of the variables are so."

3.4.1. Database setup

We have created a database using the service IONOS. Here, we create 6 fields in which we can store our variables: the id for our device, a timestamp that gets automatically filled when a new datapoint is acquired, and the variables for latitude, longitude, temperature, and movement.

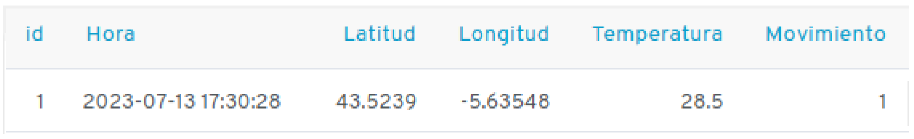
We can browse the database using Structured Query Language (SQL) to select, insert, update, and delete our data entries. Additionally, it provides PHP language examples for connecting to the database.

3.4.2. PHP document

This document, located in the Medialab server, contains the instructions required to deliver the information from the http request from Node-RED into the database. It identifies certain variables and knows where to post the corresponding values. For example, if we introduce the request of the previous example:

www.medialab-uniovi.es/post_vaquer_tracker.php?id='1'&lat='43.523933'&lon='-5.635477'&t='29.7'&m='1'

said request will access the file named “post_vaquer_tracker.php”, which contains the instructions to look for “**lat**”, “**lon**”, “**t**” and “**m**”. It also stores and identification name of the device. Additionally, it is configured in a way that stores the timestamp of the request. It then makes a connection with the database, for which it has its own password, and adds a new entry:



id	Hora	Latitud	Longitud	Temperatura	Movimiento
1	2023-07-13 17:30:28	43.5239	-5.63548	28.5	1

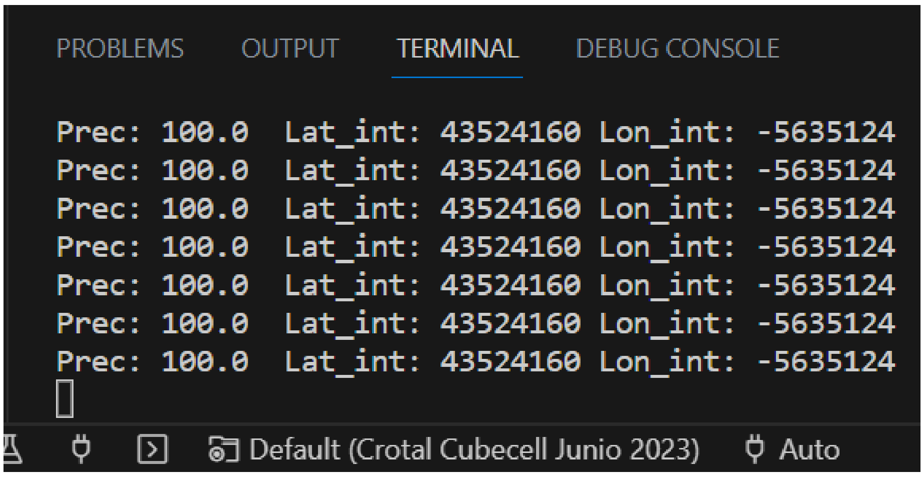
Figure 44 - Values stored in the database.

3.5. Data visualization and user interaction

At this point, we can access our database to retrieve the necessary information. This is enough for our project, but we could expand it in the future by creating a virtual dashboard using a service like Grafana.

4. Testing

While developing the prototype, all the data for the individual components (Accelerometer, GPS and LoRa message) was shown on the serial monitor of Visual Studio Code, requiring the device to be connected to the computer at all times.

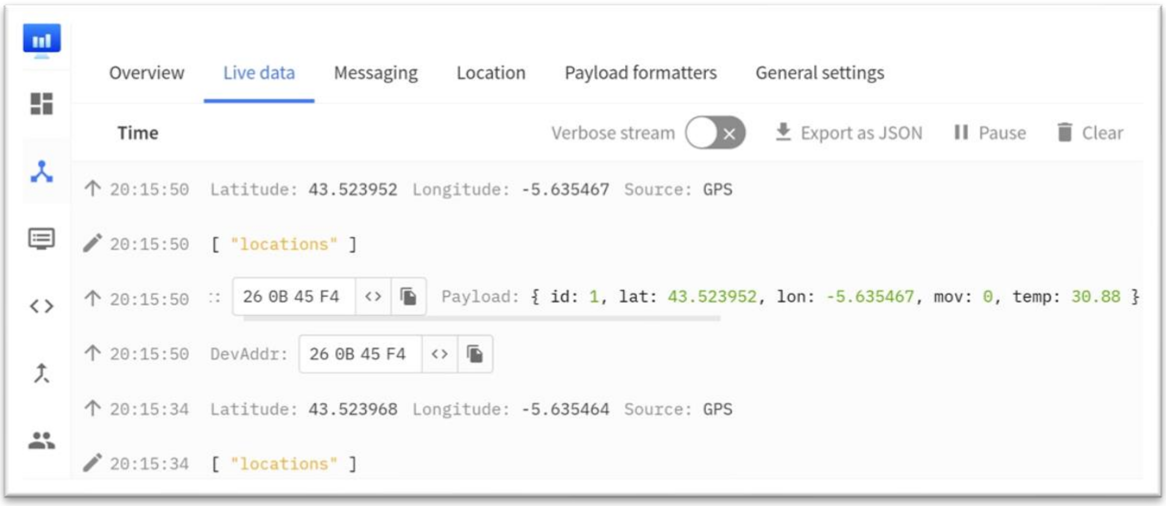


```
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
Prec: 100.0  Lat_int: 43524160 Lon_int: -5635124
```

Default (Crotal Cubecell Junio 2023) Auto

Figure 45 - Serial monitor of Visual Studio Code showing accelerometer data.

Once we had our device connected to TTN, we could establish a connection and under the “Live Data” section check that everything was working correctly. This has the advantage of having the assurance that every part of the system is working correctly, but the disadvantage of having to wait entire minutes every time we uploaded new code to the Cubecell and had to reconnect to TTN.



```
↑ 20:15:50  Latitude: 43.523952  Longitude: -5.635467  Source: GPS
✎ 20:15:50  [ "locations" ]
↑ 20:15:50  :: 26 0B 45 F4  Payload: { id: 1, lat: 43.523952, lon: -5.635467, mov: 0, temp: 30.88 }
↑ 20:15:50  DevAddr: 26 0B 45 F4
↑ 20:15:34  Latitude: 43.523968  Longitude: -5.635464  Source: GPS
✎ 20:15:34  [ "locations" ]
```

Figure 46 - Data obtained in TTN

4.1. MPU6050

The tests for the accelerometer and gyroscope module were easy: once the library was configured correctly and the CubeCell enabled the use of the module, data started showing on screen. We started using the simplest examples for the MPU6050 library, showing the constant readings on screen, and monitoring the 6 axes of the device.

When we had to replace the Arduino with the CubeCell development board, the library was not compatible anymore. Although the CubeCell is supposed to be fully compatible with Arduino, the readings of the accelerometer were wrong. To solve this, we changed the library to “Adafruit MPU6050”, by Adafruit, instead of the one provided by ElectronicCats.

This new library has many more functions to use, even enabling high-pass filters for motion detection. The increment in memory of the microcontroller allows us to use more complex and heavier functions, and so we stopped calculating differences in acceleration and rotation one for one in the code. The new configurable filter functions are much better at that and offer very reliable data.

However, problems started to arise when we interrupted the connection to its power source. If we sent the microcontroller to sleep and disabled the external voltage output pin, when we enabled it again the microcontroller would not establish a connection. Not even by calling the initializing function again would the module work. It is for this reason that we decided to use the VDD pin, which is always on, instead of the convenient Vext to manage the power consumption of the accelerometer.

4.2. GPS module Neo-6M

The GPS tests have caused many limitations when developing the systems. When coding the device, we did so inside a building, where GPS coverage is not ideal.

Despite sitting next to a window and placing the prototype on the balcony, on many occasions the data would take too long to show up, or not show up at all. Having to go out with all the equipment to sit outdoors, and many times with the glare of the sun obfuscating the light from the screen, is not an experience we would recommend. Luckily, the building is surrounded by open space. On the occasions when we brought the device home and tested it in the middle of the city, not a single correct GPS fix was established.

This would not be an issue, as the intended use of the device is on an open field outdoors, but has made development slower and more annoying.

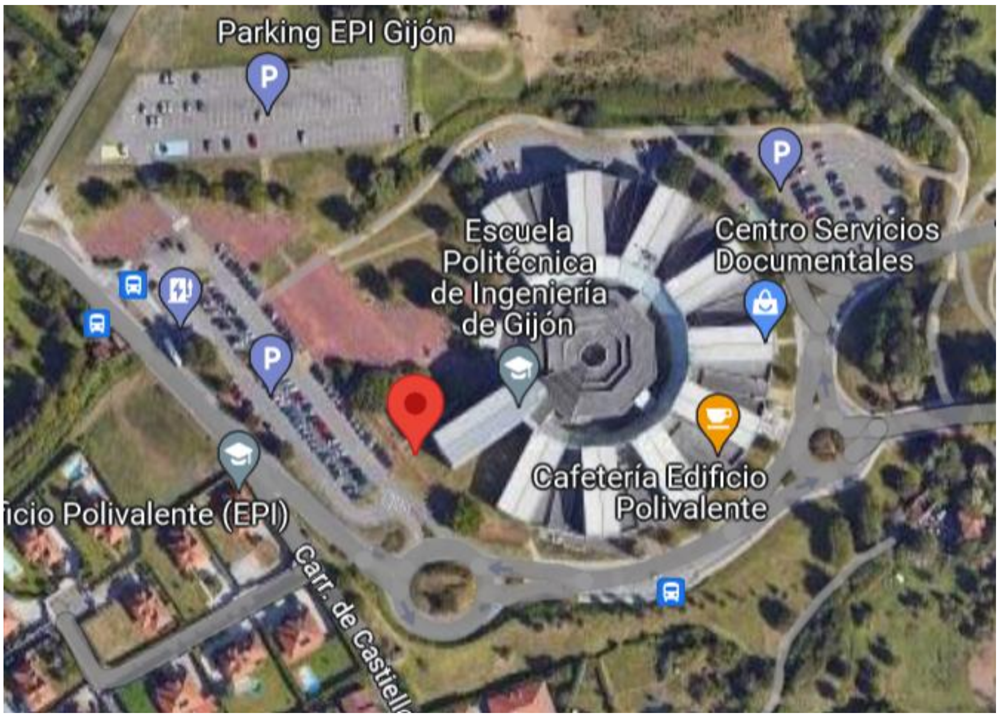

Parking EPI Gijón
Escuela Politécnica de Ingeniería de Gijón
Centro Servicios Documentales
Cafetería Edificio Polivalente
icio Polivalente (EPI)
Carr. de Castiello

Figure 47 – Lab's building and outdoor bench where tests were performed.

4.3. LoRa

The CubeCell is a wonderful way of implementing a LoRaWAN device, as it already has its own antenna and communication libraries. Additionally, it is a device present in the TTN devices catalogue, so configuring the device in TTN is as simple as making a series of clicks. While the Arduino board + RFM95 module was harder to connect to TTN, the CubeCell board established connection automatically once the keys were copied into the TTNvalues.h codes and the first test code was uploaded. However, now we need to talk about its limitations: the fact that the device requires almost a direct line of sight to the antenna.

The antenna located in the building is at one window in the Medialab department, two stories high. If we send data from Medialab or outside in the open field, there is no problem. If we go to other parts of the building, such as a different floor or walking around, we lose connection. If we walk towards the city center, the moment we get next to other buildings the connection is lost again.

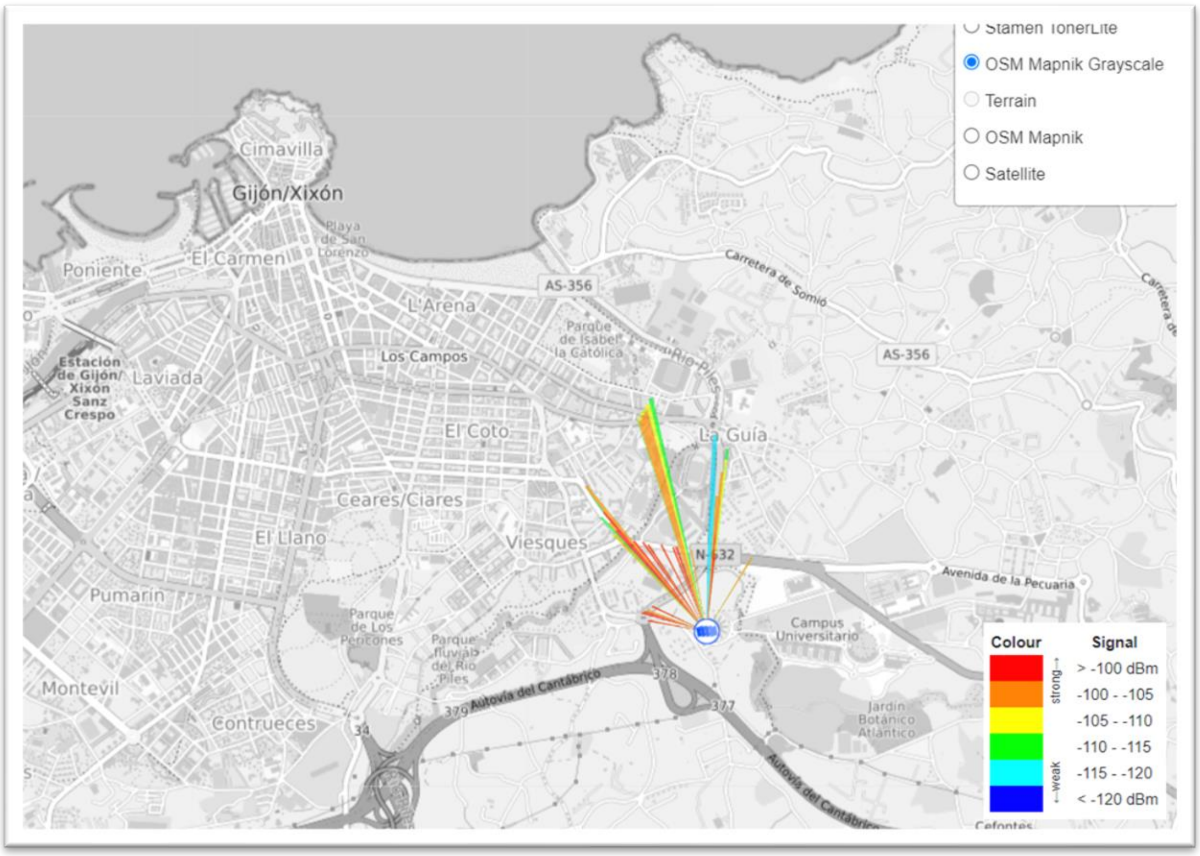


Figure 48 - Coverage of the gateway installed in Medialab.

If we take a look at the images, we can observe that most beams are directed towards one specific area, north-northwest. This is curious for an omnidirectional antenna, which should have equal distribution all around. The answer can be found in the following figure, where we can see the precise location of the antenna: at a window facing north-northwest. Only two of the beams manage to escape "through" the building, and the one going south may be coincidental, as the room is full of windows. The ideal location of such antenna would be on the roof of the building instead of on a window, but at the moment we lack the authorization to move the gateway to a more proper location.

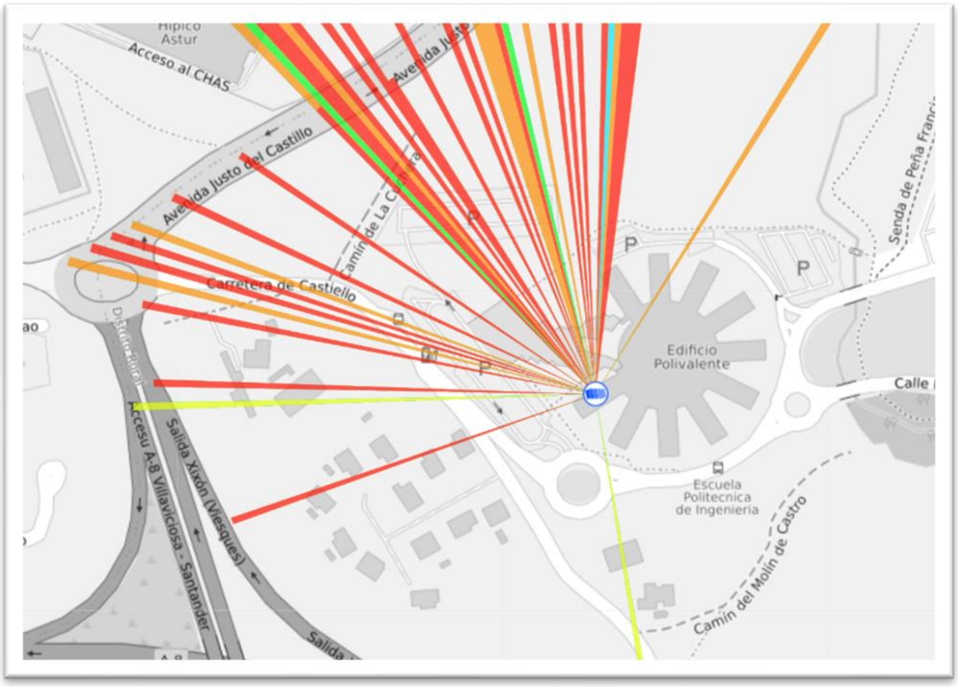
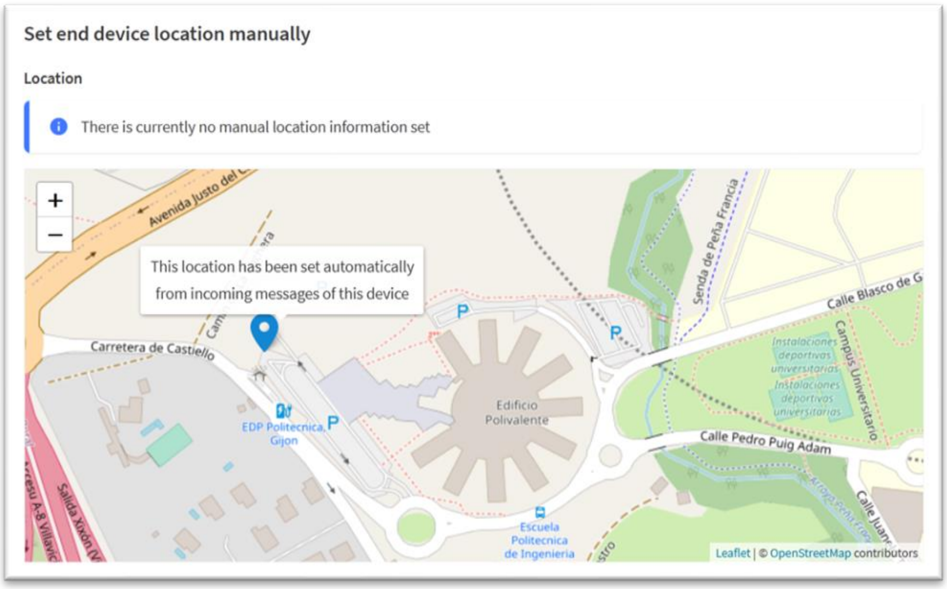


Figure 49 - Detail of the gateway's location

The lack of coverage will not be an issue for an antenna located on an open field, but the gateway we worked with had these limitations. We could work in and around the university, but at the moment we got away from it the messages stopped sending. As a reminder, debugging and visualizing data via the serial monitor in Visual Studio Code was always an option, but receiving the data in TTN was always a more complete test.


Set end device location manually

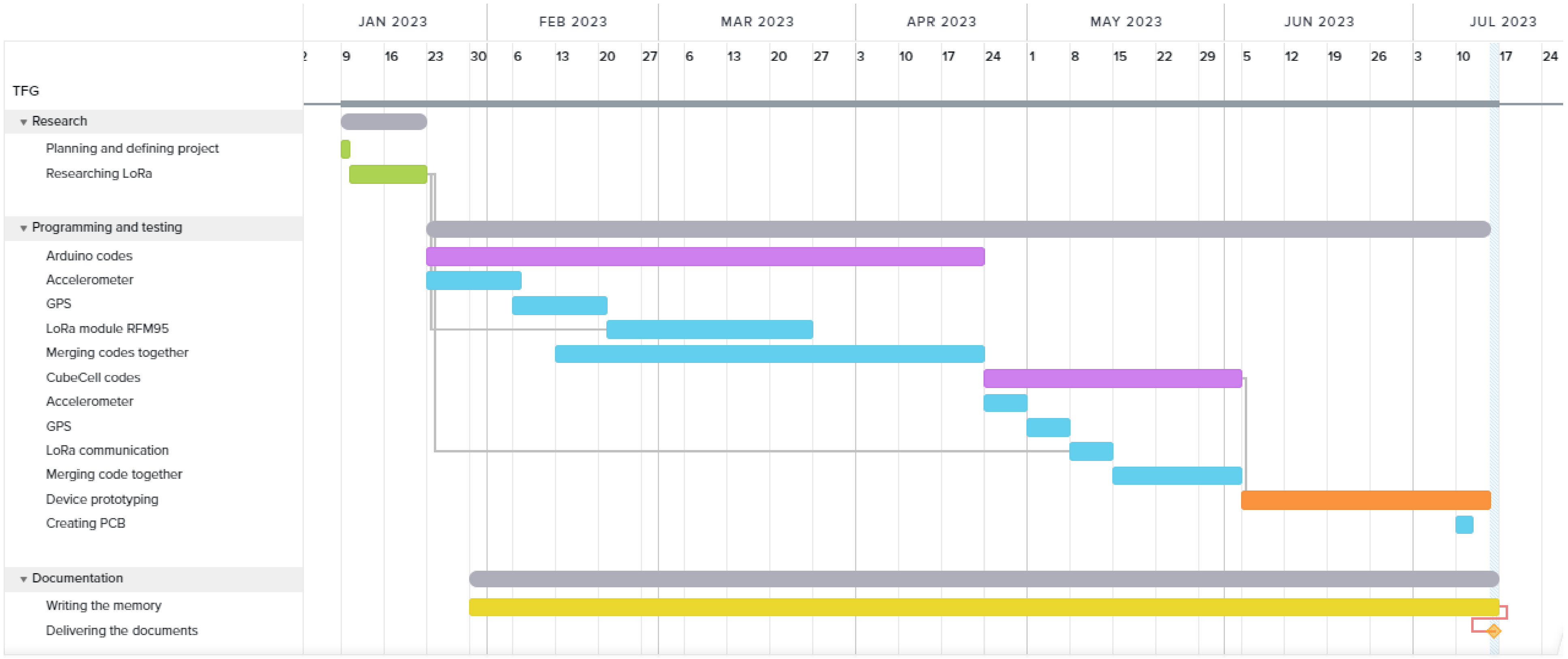
Location

There is currently no manual location information set

This location has been set automatically from incoming messages of this device

Figure 50 - GPS location sent to TTN while testing away from the lab.

5. Planning


JAN 2023
FEB 2023
MAR 2023
APR 2023
MAY 2023
JUN 2023
JUL 2023
2 9 16 23 30 6 13 20 27 6 13 20 27 3 10 17 24 1 8 15 22 29 5 12 19 26 3 10 17 24
TFG
Research
Planning and defining project
Researching LoRa
Programming and testing
Arduino codes
Accelerometer
GPS
LoRa module RFM95
Merging codes together
CubeCell codes
Accelerometer
GPS
LoRa communication
Merging code together
Device prototyping
Creating PCB
Documentation
Writing the memory
Delivering the documents

6. Results and discussion

The development of this device has been at times complex and frustrating, due to the many times we needed to change components, wait for the arrival of new ones, study the compatibility of the libraries, develop new code, and rewrite it.

Additionally, there are many aspects of this project that we hadn't studied in our career path as electronic engineers, such as telecommunication techniques or server scripting languages like PHP. The novelty of these systems added some layers of complexity to the project.

However, the result is a working prototype that manages to send all the data we requested to the database using LoRa communication. For an electronic engineer, this is a satisfactory result after all the hard work.

6.1. Limitations

Most of the limitations of this device arise from the local availability of LoRa networks to connect to. In the following figure we observe the LoRa network around Asturias. It is worth mentioning the range that LoRa antennas can achieve when there are no obstacles around, such as the one connecting Santander and Ribadesella 100 km away.

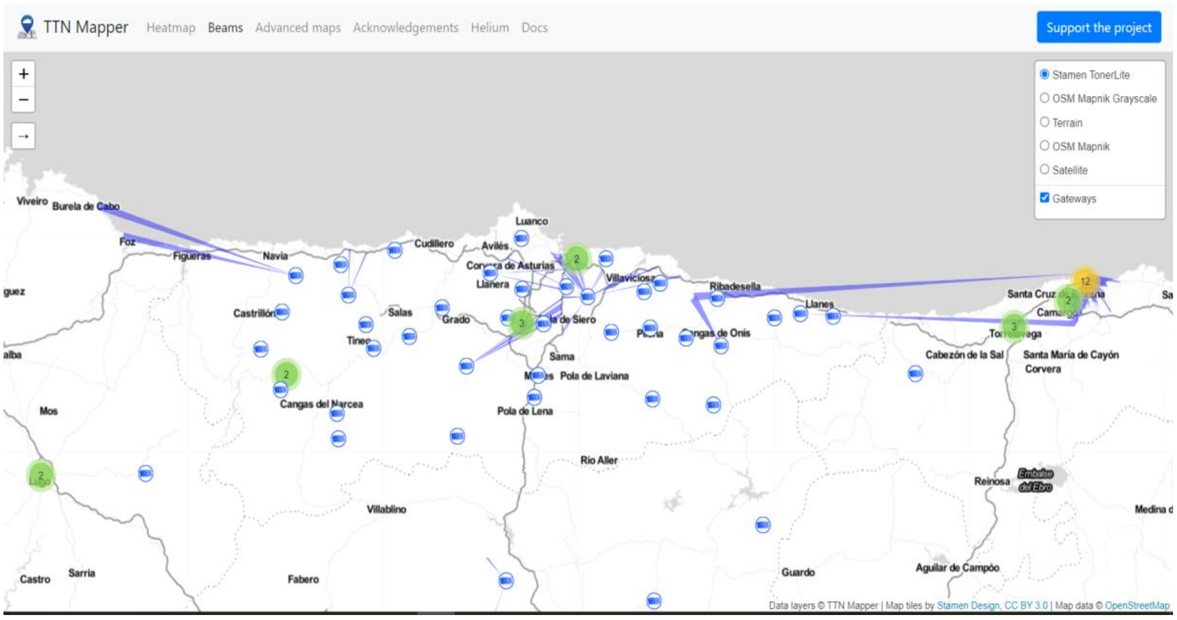


Figure 51 - LoRa network in Asturias and neighboring regions.

Along the development of this project, we have seen a large increase of LoRa gateways in Asturias from community-based organizations. Additionally, the principality of Asturias will install new gateways in old digital television towers, expanding the network especially for the more isolated rural areas (9). Our device will be limited until that broader infrastructure is built. As seen in the previous image, the current coverage is less than ideal. Asturias presents a harsh orography, with countless mountains and valleys that difficult the requisite of having direct line of sight to the antenna.

While LoRa is an ideal technology for IoT applications, the circumstances of our region difficult the use of the device, which at the moment would require installing a gateway on the field where the animals roam. However, for regions with a flat terrain, this would be a very effective technology to use. It is no surprise that the Netherlands is the most populated area in terms of LoRa gateways and applications.

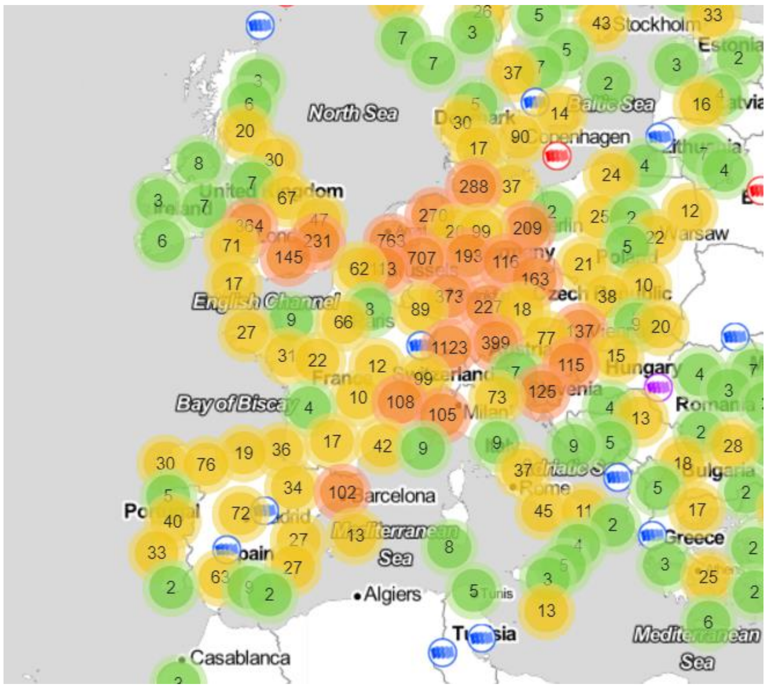


Figure 52 - Map of LoRa gateways in Europe.

It is also our intent that, using this document as a guide, anybody could replicate our device by themselves. It is also one of the working principles of Medialab to share our knowledge and ideas to the public. While our geographical circumstances may not be ideal and the infrastructure not fully developed yet, it is our hope that anybody can make their own device based on our work and put it to use in better conditions.

6.2. Future improvements

As of the day we finish this document, the Printed Circuit Board we designed has not arrived yet, and thus we have worked only with the components on a protoboard. Our intention is to mount all of the modules in a fully functional PCB and finish the prototype by designing a case. This "shell" for the device will have an access for the temperature sensor of the accelerometer to be in contact with the neck of the animal, and two openings at the top, one for a solar panel to be mounted and another for the ceramic antenna of the GPS to be exposed. As the device is intended for outdoor use permanently by an animal, we will need to waterproof the components as well.

When the PCB and custom shell are delivered, we will then have a field test with an actual cow, sending real data to be studied. Up to date, the device has been tested while strapped to an arm, but the real use in the field will prove an enlightening experience and will symbolize having a real product instead of a working prototype.

Another interesting development is the discovery of a new development board: CubeCell – GPS-6502 by Heltec Automation (21). This module already has a GPS antenna incorporated and a retail price of around 27 €. Our device with the GPS module and the old Cubecell is slightly cheaper, but this alternative device would be even simpler in terms of number of modules. For future integrations of this project, we will probably use these new devices.

6.3. Conclussion

This project has been a huge learning experience that has vastly extended my previous knowledge as an electronic engineering student. While learning so many new things for a single project has been overwhelming and frustrating at times, I am satisfied by the result, and grateful to have been able to work in such an enriching environment as Medialab.

7. References

(1) Online magazine for veterinary RumiNews (2022, January 19) https://rumiantes.com/las-ventajas-de-la-geolocalizacion-de-los-rebanos-de-rumiantes/

(2) La Voz de Asturias. (2021, June 5). ¿Cuánto se paga por los daños del lobo por cabeza y ganado? https://www.lavozdeasturias.es/noticia/asturias/2021/06/04/paga-danos-lobo-cabeza-ganado/00031622806121301381977.htm

(3) Virtual geofencing company Nofence (n.d.) https://www.nofence.no/en/

(4) IndiaNIC (2021, June 24) Livestock Tracking And Geofencing Solution For Agriculture Industry https://www.indianic.com/blog/automation/geofencing-in-agriculture-industry-for-livestock-tracking.html

(5) ZERVAS, G. (1998), Quantifying and optimizing grazing regimes in Greek mountain systems. Journal of Applied Ecology, 35: 983-986. https://doi.org/10.1111/j.1365-2664.1998.tb00019.x

(6) Unold O et al. (2020) IoT-Based Cow Health Monitoring System, Computational Science – ICCS 2020 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7302546/

(7) Cowmanager n.d.) https://www.cowmanager.com/cow-management/modules/health/

(8) El reto de la geolocalización en la ganadería (2021, June 7) https://blog.orange.es/innovacion/geolocalizacion-ganaderia/

(9) Smartlightning. (n.d.). Asturias desplegará una red inalámbrica LoRa para impulsar la conectividad y el IoT en la región . https://smart-lighting.es/asturias-red-inalambrica-lora-iot/

(10) Transparencia.asturias.es (2021, October 11) El Principado desplegará una red inalámbrica que facilitará la comunicación e intercambio de datos entre dispositivos electrónicos transparencia.asturias.es/web/asturias/detalle-noticia

(11) Cellnex (2023, February 27). El Principado presenta un acuerdo con Cellnex para implantar proyectos piloto de internet de las cosas en el ámbito rural asturiano https://www.cellnex.com/es-es/noticias/principado-acuerdo-cellnex-implantar-proyectos-piloto-internet-cosas-ambito-rural-asturias/

(12) Wattson, J. (n.d.). MEMS Gyroscope Provides Precision Inertial Sensing in Harsh, High Temperature Environments | Analog Devices. Retrieved from https://www.analog.com/en/technical-articles/mems-gyroscope-provides-precision-inertial-sensing.html

(13) The Things Network | The Things Stack for LoRaWAN. (n.d.). https://www.thethingsindustries.com/docs/getting-started/ttn/

(14) La voz de Asturias (2022 june 14) https://www.lavozdeasturias.es/noticia/asturias/2022/06/13/asturias-registra-800-ataques-prohibio-caza-lobo/00031655134943943743377.htm

(15) GIS Geography (2022 May 31) How GPS Receivers work https://gisgeography.com/trilateration-triangulation-gps/

(16) GPS.gov (n.d.) Official U.S. government information about the GPS https://www.gps.gov/systems/gps/performance/accuracy/

(17) RF Wireless World (n.d.) NMEA Sentences https://www.rfwireless-world.com/Terminology/GPS-sentences-or-NMEA-sentences.html

(18) Image: heart rate sensor mounted on a belt for cows https://www.researchgate.net/figure/Heart-rate-sensors-and-receiver-mounted-on-cow_fig2_267987403

(19) Elektor Store (n.d.) link to pulsimeter device https://www.elektor.com/sparkfun-pulse-oximeter-and-heart-rate-sensor-max30101-max32664-qwiic

(20) Node-RED (n.d) Low-code programming for event-driven applications https://nodered.org/#:~:text=Node%2DRED%20is%20a%20programming,runtime%20in%20a%20single%2Dclick.

(21) Heltec Automation (n.d.) CubeCell GPS-6502 https://heltec.org/project/htcc-ab02s/