UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

DISEÑO DE UN SENSOR DE TEMPERATURA IOT PARA LA RED LORA

DESIGN OF IOT TEMPERATURE SENSOR FOR LORA NETWORK

Autor:

Rodríguez Lallana, Mario

Marta Herráez Sánchez

Vilnius Gediminas Technikos Universitetas

Valladolid, Junio de 2018

TFG REALIZADO EN PROGRAMA DE INTERCAMBIO

TÍTULO: DESIGN OF IOT TEMPERATURE SENSOR FOR LORA NETWORK / DISEÑO DE UN SENSOR DE TEMPERATURA IOT PARA LA RED LORA

ALUMNO: Mario Rodríguez Lallana

FECHA: 07/06/2018

CENTRO: Vilnius Gediminas Technikos Universitetas

TUTOR: Dainius Udris

Resumen:

El objetivo de este proyecto es implementar una solución válida para un proyecto de IoT basado en un LM35 (sensor de temperatura), que proporciona los datos que se transmitirán a través de una WAN LoRa, y un módulo, el TTGO LoRa Wifi V1. Esté módulo está formado por: el chip ESP32, un OLED controlado por el SSD1306 y un transceptor LoRa controlada por el SX1276.

La transmisión de RF tendrá lugar en la banda de la UE reservada para la tecnología LoRa, 868 MHz. Y los paquetes LoRa WAN serán enviados a un Gateway LoRa que los redirigirá a la nube a nuestra aplicación en línea.

Después de todo, esta aplicación, que se hará en línea con la plataforma TheThingsNetwork.org, tendrá el deber de gestionar el paquete que recibe decodificarlos y proporcionar un entorno fácil de usar para mostrar estos datos para nosotros, y la información de la misma.

Palabras clave:

Sensor de temperatura, Internet de las Cosas, tecnología LoRa, LoRa network, ESP32

Vilnius Gediminas Technical University

Faculty of Electronics

Mario Rodríguez Lallana

Design of IoT temperature sensor for Lora network

Bachelor`s Thesis

Supervisor: Dainius Udris,

Department of Electrical Engineering

Vilnius 2018

Declaration: I hereby declare that this Bachelor's thesis, my original investigation and achievement, submitted for the Bachelor's degree at Vilnius Gediminas Technical University has not been submitted for any degree or examination.

Mario Rodríguez Lallana

Date ……………….................

Signature ……………….................

INDEX OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

1.INTRODUCTION.

What is the IoT?

The Internet of Things [1] is a concept that is based on the interconnection of any product with any other product around it. From a book to the fridge in your own home. The aim is to make all these devices communicate with each other and thus become more intelligent and independent.

Its significance can be brutal in both the economic and social spheres. Even greater than the digital age. The Internet of Things allows computers to interact with real-life elements and gain independence from human beings, leaving us in charge of what really matters.

What is LoRa?

IoT devices are normally designed to have a long service life ranging from five to eight years. This requires continuous support of the chosen network connectivity throughout the life of the device. Due to the uncertainty regarding the longevity of the 2G network, designers should consider alternative connectivity that not only has long-term secure support, but also fits the power consumption, communication range and low cost limitations used with typical IoT applications.

Among the potential candidates, LoRa technology has strong technical credentials and is already being used in applications that require reliable communications over distances of several kilometers, such as wireless meter reading and street lighting control. It is a sub-GHz Low-Power Wide Area Network (LPWAN) technology that maintains a data rate of 0.3 kbps to 50 kbps depending on the range and duration of the message. Transmission distances can be up to 15 or 20 km.

1.1. Goals:

The aim of this project is to implement a valid solution for an IoT project based on a LM35 (temperature sensor), that provides the data which will be transmitted through a LoRa WAN, and a LoRa technology board, the TTGO LoRa Wifi V1. This board will be ruled by the ESP32 chip, this one will control: an OLED display to have a better control of the state of our transmission process through the SSD1306 controller and a LoRa technology transceiver with an antenna controlled by the SX1276. For each controller we will need different libraries to perform an optimal communication between them.

The RF transmission will take place on the EU band reserved for the LoRa technology, 868 MHz. And the LoRa WAN packages will be sended to a LoRa gateway which will redirect them to the cloud to our online application.

After all, this application, that will be made online with the TheThingsNetwork.org platform, will have the duty to manage the package that it receives decode them and provide a user-friendly environment to display this data for us, and the information of it.

2.ANALYTICAL PART.

2.1. Structure and characteristics of Lora network.

2.1.1. LoRa Network architecture

LoRa [2], which stands for "Long Range", is a long-range wireless communications system, promoted by the LoRa Alliance. This system aims at being usable in long-lived battery-powered devices, where the energy consumption is of paramount importance. LoRa can commonly refer to two distinct layers:

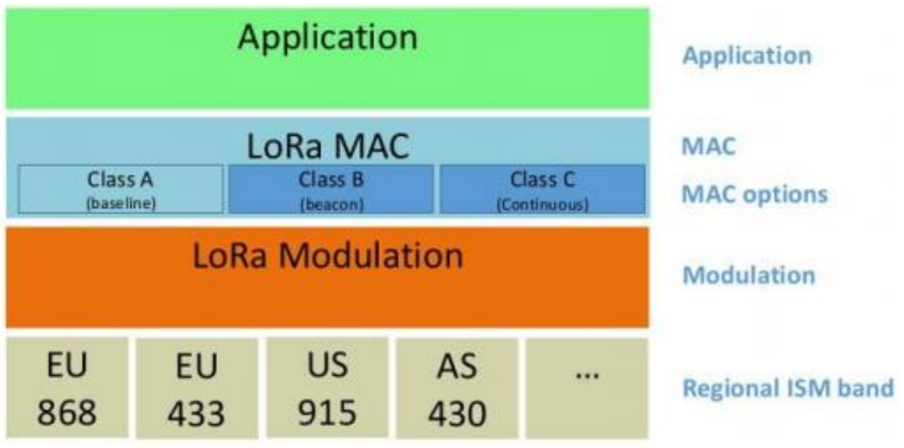


Figure 1 LoRa WAN architecture and devices classes [7]

(i) a physical layer using the Chirp Spread Spectrum (CSS) radio modulation technique; and Sensors 2016, 16, 1466 4 of 18

(ii) a MAC layer protocol (LoRaWAN), although the LoRa communications system also implies a specific access network architecture.

Many existing deployed networks utilize a mesh network architecture. In a mesh network, the individual end-nodes forward the information of other nodes to increase the communication range and cell size of the network. While this increases the range, it also adds complexity, reduces network capacity, and reduces battery lifetime as nodes receive and forward information from other nodes that is likely irrelevant for them. Long range star architecture makes the most sense for preserving battery lifetime when long-range connectivity can be achieved.

The basic architecture of a LoRaWAN network is as follows: end-devices communicate with gateways using LoRa with LoRaWAN [3]. Gateways forward raw LoRaWAN frames from devices to a network server over a backhaul interface with a higher level of performance, typically Ethernet or 3G.

Consequently, gateways are only bidirectional relays, or protocol converters, with the network server being responsible for decoding the packets sent by the devices and generating the packets that should be sent back to the devices. There are three classes of LoRa end-devices, which differ only with regards to the downlink scheduling.

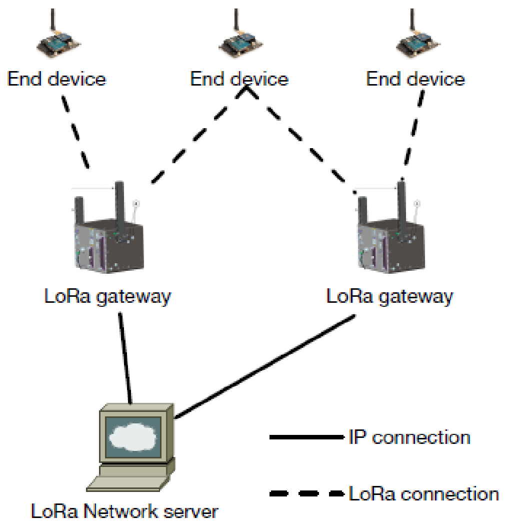


Figure 2 LoRa network architecture [6]

2.1.2. LoRa Physical Layer

The LoRa physical layer [4], developed by Semtech, allows for long-range, low-power and low-throughput communications. It operates on the 433-, 868- or 915-MHz ISM bands, depending on the region in which it is deployed (868 MHz in Europe). The payload of each transmission can range from 2–255 octets, and the data rate can reach up to 50 Kbps when channel aggregation is employed. The modulation technique is a proprietary technology from Semtech.

LoRa is a chirp spread spectrum modulation, which uses frequency chirps with a linear variation of frequency over time in order to encode information. Because of the linearity of the chirp pulses, frequency offsets between the receiver and the transmitter are equivalent to timing offsets, easily eliminated in the decoder.

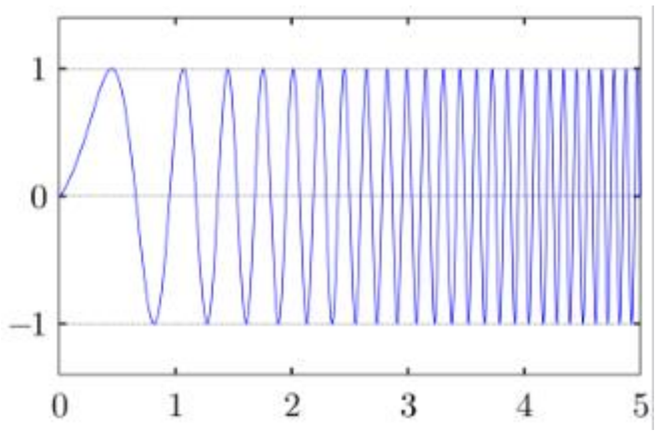


Figure 3 Up-Chirp spread spectrum modulation example in time domain [6]

To figure out the chirp modulation [5] works deeply, let’s take a look at a commonly used frequency modulation techniques, the frequency shift keying, in which the frequency switch between just two values, like a digital signal.

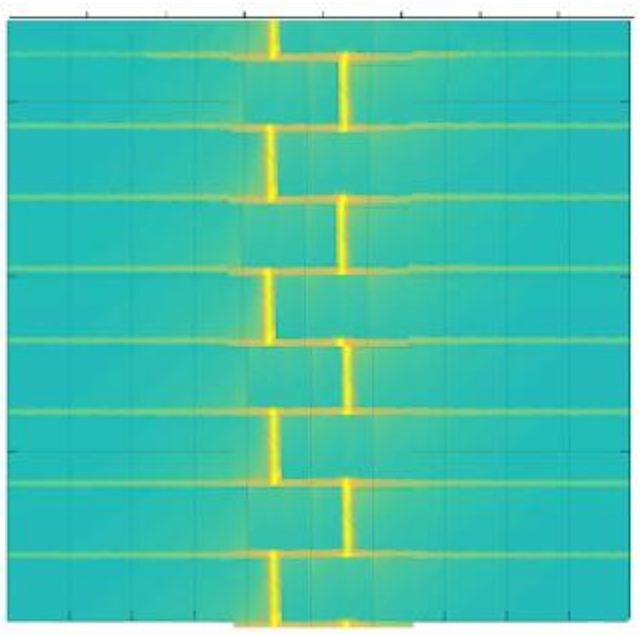


Figure 4: example of FSK frequency shift keying. [5]

And on the other hand, we have a chirp spectrum example in which we can see easily how the frequency takes a lot of different values from its bandwidth making lines called: upchirps and downchirps.

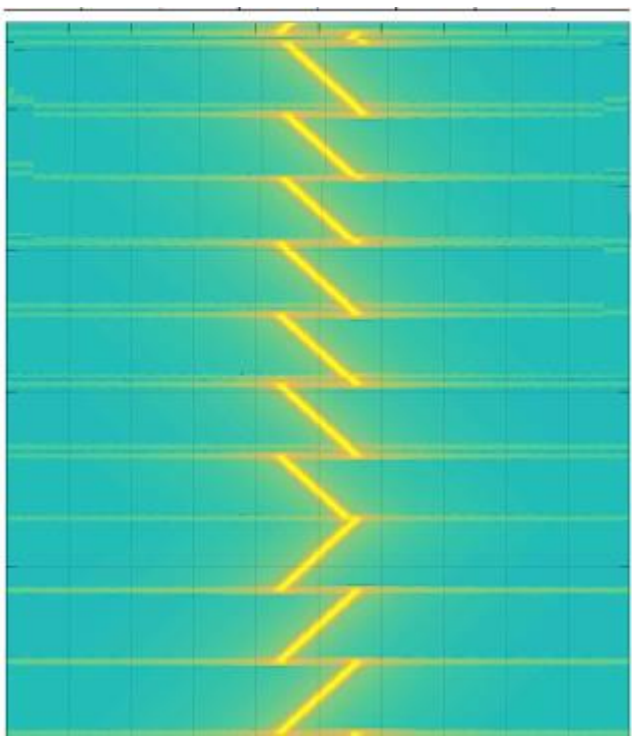


Figure 5: example of chirp modulation [5]

The direction, bandwidth use and speed of growing slope (sweep rate), are the two parameters that will give us the possibility to send a lot of data with smaller frames than other modulation techniques.

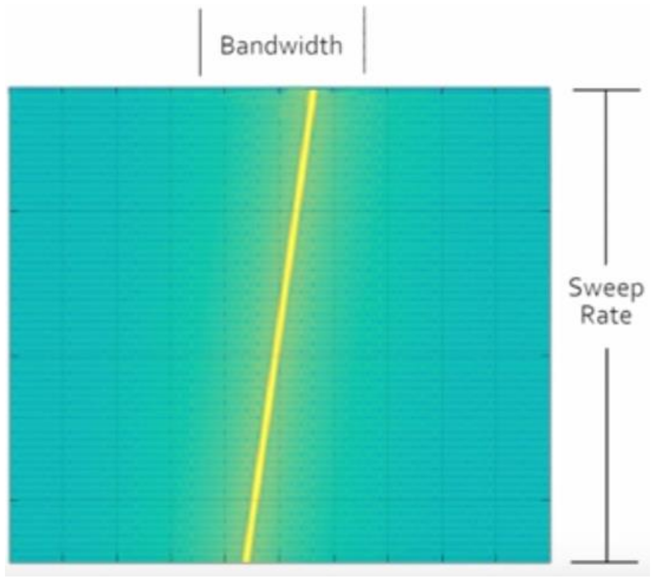


Figure 6: Bandwidth and Sweep rate in the chirp modulation [5]

Physical Frame Format

In the physical layer [6] of our LoRa WAN network, we find a frame format to encapsulate our payload, and add all the physical layer context information. This format is normally a preamble and the bare payload, but this format may change with two additional but optional frame fields: a heading and a CRC (Cyclic redundancy check).

- Preamble:

The preamble starts with a sequence of constant upchirps that cover the whole frequency band of a given bandwidth. The last two upchirps of this preamble encode the *sync word*. The *sync word* is a one-byte (8-bits) value that is used to pair LoRa Networks which use the same frequency band. For it, devices configured with a given *sync word* will stop listening to a transmission if the decoded sync word does not match its configuration (if they don´t have the same sync word). The *sync word* is followed by two and a quarter downchirps, for a duration of 2.25 symbols. The total duration of this preamble can be configured between 10.25 and 65,539.25 symbols.

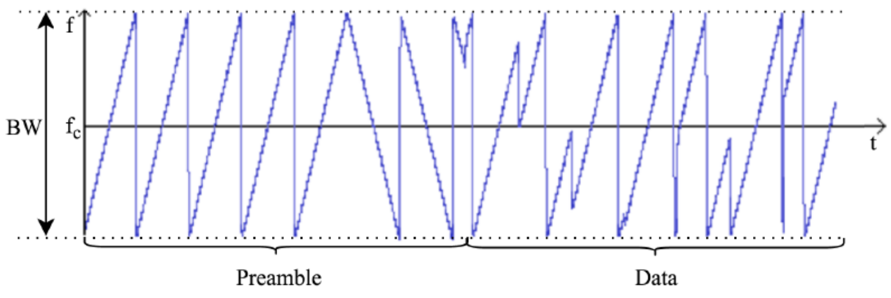
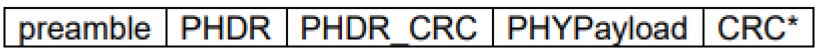


Figure 7 preamble of a physical frame format [6]

- Optional header:

After the preamble we have the possibility of adding an optional heading. If we use it, it´s transmitted with a code rate of 4/8 (if the code rate is k/n, for every k bits of useful information, the coder generates a total of n bits of data, of which n-k are redundant). That says the size of a payload (in bytes), the code rate used for the end of transmission and whether or not a 16-bit CRC for the payload is present at the end of the frame. In addition, the heading includes a CRC to help and give the power of discarding packets with invalid headers to the receiver. To store the payload size is stored using 1 byte, so the limit of the payload is 255 bytes. The header is optional to allow disabling it in situations where it is not necessary, for instance when the payload length, coding rate and CRC presence are known in advance.



preamble	PHDR	PHDR_CRC	PHYPayload	CRC*

Figure 8 LoRa WAN Physical Layer Frame Format [7]

2.1.3. LoRa MAC layer

LoRa WAN gives us a MAC layer [7], to manage all the incoming requests of transmission from the end-devices of our network, allowing them to communicate through LoRa modulation. It uses the ISM bands (as many others radiofrequency applications), it allows everybody to create public or private IoT networks in an accessible way anywhere using the hardware and the software needed for it.

LoRaWAN's network protocol and architecture decisively determine a node's battery life, network capacity, quality of service, security, and the variety of network applications. Unlike LoRaWAN, LoRa has no rules or standards beyond the physical layer limitations offered by the devices or those imposed by the library used. LoRaWAN can use LoRa or FSK modulation at the physical level

It was designed for sensor networks where all of them would be exchanging packages, with a low data rate in long time intervals, like transmitting the temperature of an aquarium one time for day.

Another interesting fact about LoRa wan it´s that you can´t transmit directly between end-devices, the “message” need to go first to the network server and then it can go to the other end device.

LoRa network consists of several elements:

LoRa Nodes / End Points: LoRa end points are the sensors or application where sensing and control takes place. These nodes are often placed remotely. Examples, sensors, tracking devices, etc.

LoRa Gateways: Unlike cellular communication where mobile devices are associated with the serving base stations, in LoRaWAN nodes are associated with a specific gateway. Instead, any data transmitted by the node is sent to all gateways and each gateway which receives a signal transmits it to a cloud-based network server.

Typically, the gateways and network servers are connected via some backhaul (cellular, Wi-Fi, ethernet or satellite).

Network Servers: The networks server has all the intelligence. It filters the duplicate packets from different gateways, does security check, send ACKs to the gateways. In the end if a packet is intended for an application server, the network server sends the packet to the specific application server.

Using this type of network where all gateways can send the same packet to the network server, the need of hand-off or handover is removed. This is useful for asset-tracking application where items or assets move from one location to another.

-LoRa End-Device Class

In this network the gateway serve just as relays of signals from random senders packages to forward them to the network server to store it and process it. So as to that the send-devices are associated with the network server, that at the end of each transmission will be the one that recognize duplicate packets, and if a replay is needed they choose the proper gateway to send it, even to the end-device.

In LoRa WAN we have three different classes of end-devices [8] to address the various needs of each use of the network and its technology.

Like other networks, where end devices can have different capabilities depending on devices classes, end nodes in LoRa WAN network can have different device classes.

Each device class is a trade-off between network downlink communication latency versus battery-life.

- Class A (All), Bi-directional:

The pure asynchronous one, the uplink transmission is just one slot while the downlink consists of two slots. The uplink is scheduled by the end-devices based on his own demanding needs. High probability of collisions, if one node is transmitting and another wakes up and decides to transmit on the same frequency channel with similar settings, a collision will occur.

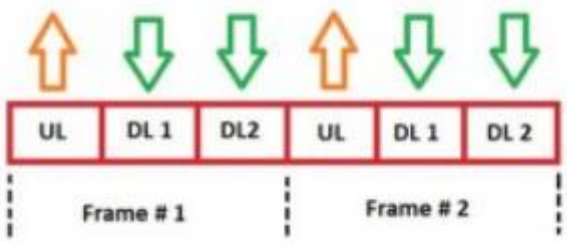


Figure 9: Class A type device transmission [8]

The downlink takes place in two receiving windows of a specific time, only after one package is sent by the end-device, it puts itself in hearing mode.

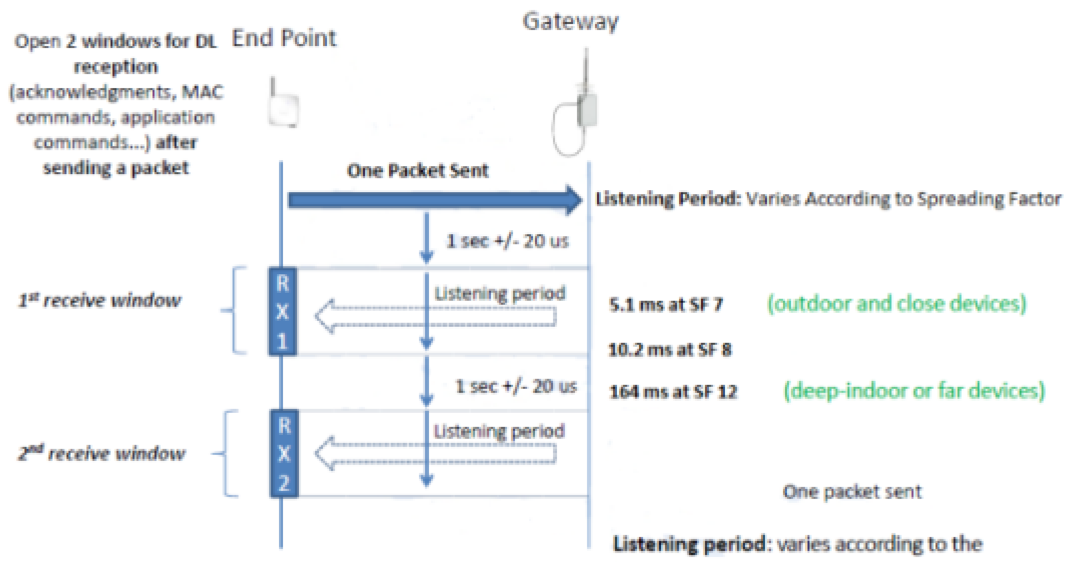

Open 2 windows for DL reception (acknowledgments, MAC commands, application commands...) after sending a packet
End Point
Gateway
One Packet Sent
Listening Period: Varies According to Spreading Factor
1 sec +/- 20 us
1st receive window
RX1
Listening period
5.1 ms at SF 7
(outdoor and close devices)
10.2 ms at SF 8
1 sec +/- 20 us
164 ms at SF 12
(deep-indoor or far devices)
2nd receive window
RX2
Listening period
One packet sent
Listening period: varies according to the

Figure 10 Class A device timing diagram [8]

This is the most energy-saving class, and it's the most used for sporadic needing communications.

- Class B (Beacon), Bi-directional with scheduled

We could have called it the one with random receiving windows, because the downlinks are planned after one uplink, when we say "planned" we are talking about a specific and selected number of reception windows and its listening duration. In general, it might be at least two downlink slots and an extra one. To start the downlink at a specific time, the device receives a time synchronized Beacon signal from the gateway.

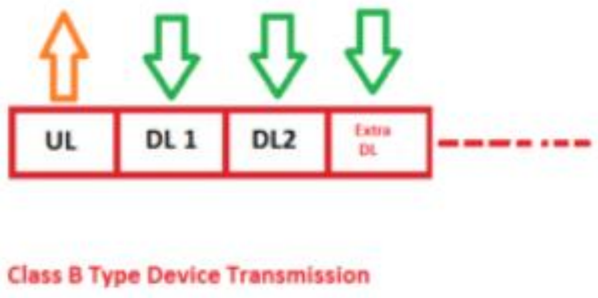


Figure 11 Class B type device transmission [8]

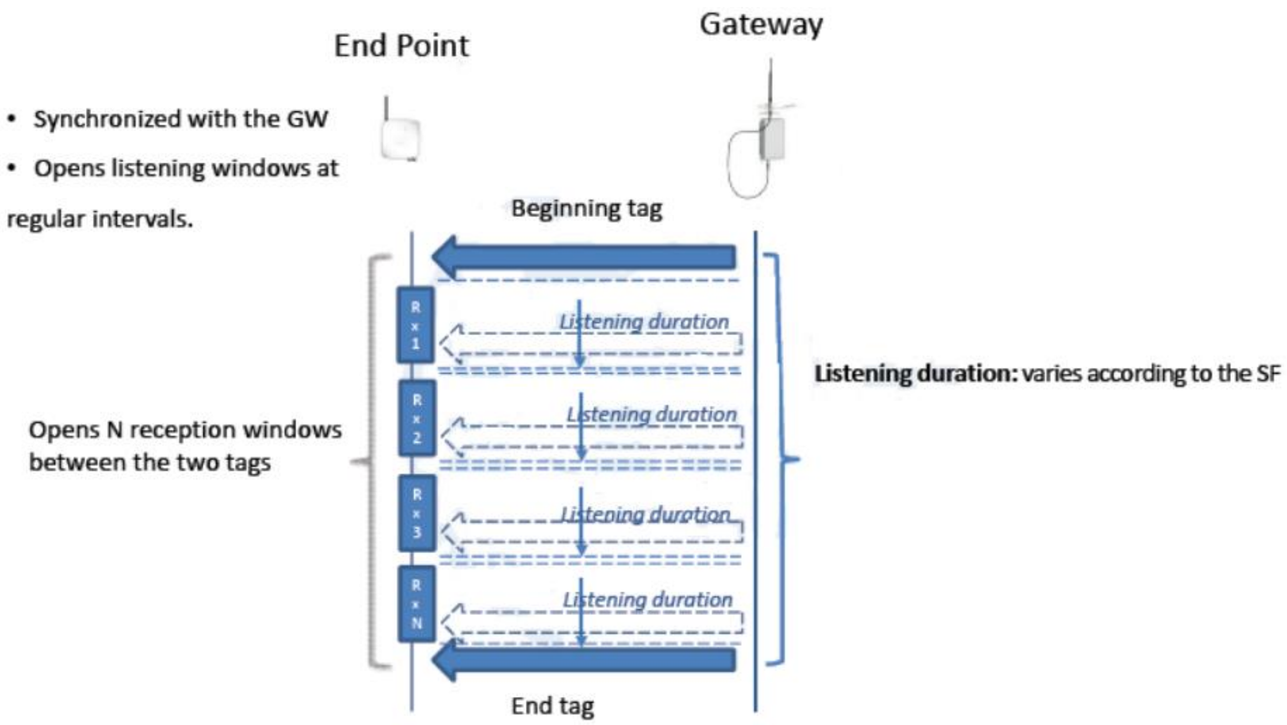


Figure 12 Class b device timing diagram [8]

- Class C (Continuous), Bi-directional with maximum receive slots

This class of devices are always ready to receive transmission, always hearing unless they want to transmit. Its easy to figure out that this is the most power consuming class out of the three, it’s a battery save-less way of performing. But these are so used for demanding applications as real time traffic management.

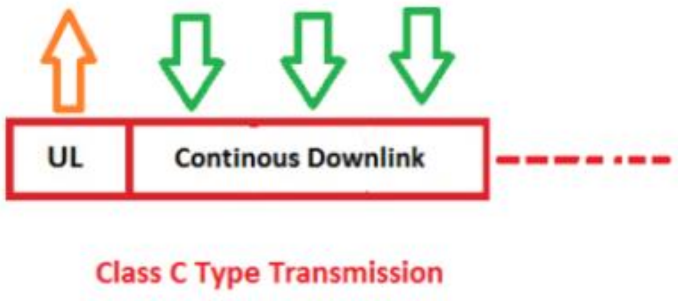


Figure 13 Class C type transmission [8]

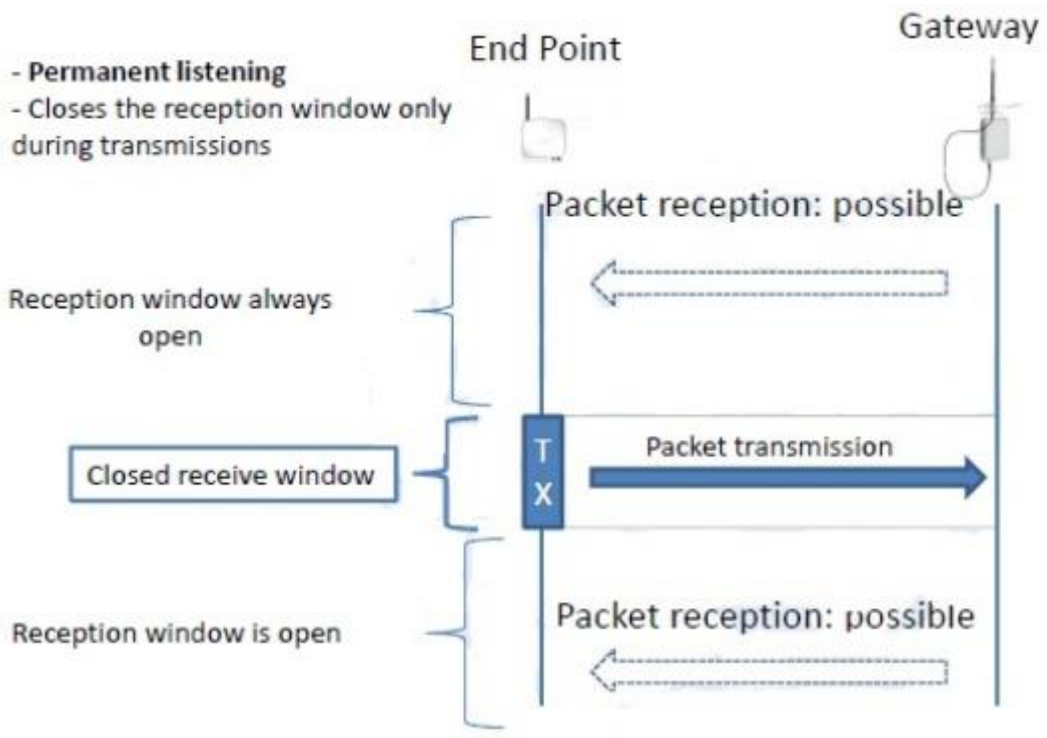

End Point
Gateway
- Permanent listening
- Closes the reception window only during transmissions
Packet reception: possible
Reception window always open
Closed receive window
TX
Packet transmission
Reception window is open
Packet reception: possible

Figure 14 Class C device timing diagram [8]

LoRa WAN MAC layer message format:

We should notice that the PHYPayload [7] its encapsulated again, it has a MAC header, the MAC Payload and a Message Integrity Code, a four-byte code calculated from the Network session key (NwkSKey)

MHDR	MACPayload	MIC[6]
1 byte	1-M bytes	4 bytes

Figure 15 PHYPayload [7]

The MAC header specifies the message type and format version of the specification of the LoRaWAN layer with which it has been encoded. There are six types of MAC messages:

MType	Descripción
000	Join Request
001	Join Accept
010	Unconfirmed Data Up
011	Unconfirmed Data Down
100	Confirmed Data Up
101	Confirmed Data Down
110	RFU[7]
111	Proprietary

Figure 16 MAC commands [7]

The MACPayload includes a frame header, optional port field and a payload optional frame.

FHDR	FPort	FRMPayload
7-23 bytes	0-1 bytes	0-N bytes

Figure 17 MAC Payload [7]

The frame header contains the address with which the device within the network, a FCtrl field to enable the Adaptive data rate, a frame counter and a FOpts field in case you want to transmit a MAC command. The FPort field is used to determine whether the field FRMPayload contains MAC commands or application data as a measurement.

DevAddr	FCtrl	FCnt	FOpts
4 bytes	1 byte	2 bytes	0-15 bytes

Figure 18 detailed FHDR [7]

The seven most significant bits of the DevAddr field are used for the network identifier (NwkID) while the remaining twenty-five correspond to the network address (NwkAddr), which can be assigned by the network administrator.

Maximum MACPayload field size varies by frequency band the data rate and the absence of the control field (FOpts):

DataRate	M (bytes)	N (bytes)
0	59	51
1	59	51
2	59	51
3	123	115
4	230	222
5	230	222
6	230	222
7	230	222

Figure 19 MAC payload sizes varying with the frequency [7]

Security in LoRa WAN:

In some scenarios, security is indispensable and necessary when dealing with data from an entire city, which may contain information that needs to be protected. That's why LoRaWAN incorporates several layers of encryption [9], with their corresponding keys, that make use of the AES128 encryption algorithm to protect communications:

- **Network Session Key**: 128-bit key that guarantees security to network level.
- **Application Session Key**: 128-bit key that ensures security end-to-end application level.
- **Application Key**: 128-bit key that guarantees extreme security to application level, used only in OTAA.

LoRaWAN, being a standardized protocol, has the following features set out in your specification, allowing the interoperability between the devices implementing this technology, regardless of the external libraries used. The transmission power of the RN2483 modules, the structure of the LoRaWAN packets and the communication between nodes in a LoRaWAN network are presented below.

My Network:

A little diagram of our particular case IoT network:

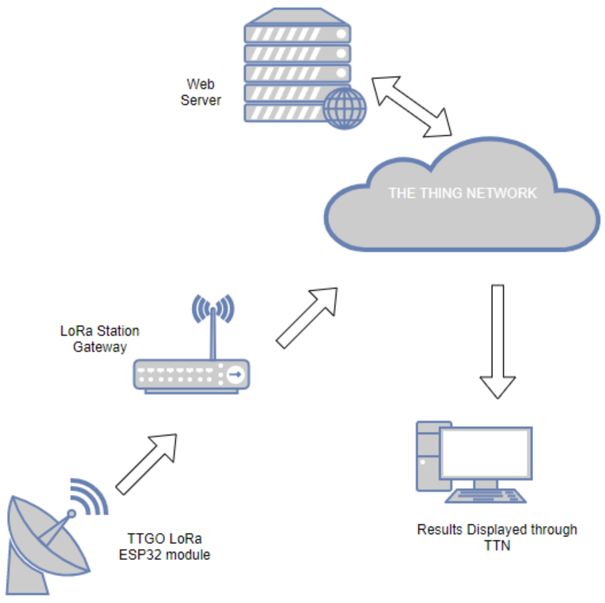


Figure 20 my network

2.2. Lora endpoint realization examples, chipsets and their comparison.

Popular ESP32 LoRa boards

The following boards are popular for prototyping because they combine an ESP32, a LoRa tranceiver, an OLED 128x64 display:

- Heltec Wifi LoRa 32
- TTGO LoRa32

These boards come in different versions and there are separate versions for 433/470MHz and 868/915MHz.

This boards are supposed to bring a Li-ion battery, but in my case it doesn´t.

LoRa antenna: external, connected via cable with I-PEX connector; 868MHz have 5cm external whip antenna with SMA connector; 433MHz have a helical wire antenna. WiFi/Bluetooth antenna: single on-board.

There is a dedicated LED *(non-programmable)* for the battery. It is on when the battery is charged. When no battery is connected: it is off when powered via 3.3V pin but flashes when powered via USB or 5V pin.
A second LED (white on Heltec, blue on TTGO) *(programmable)* is connected to pin 25, on TTGO V2 the LED is connected to pin 22 but it is useless (see below).

LoRa Performance

The LoRa performance of the Heltec and TTGO boards has shown to be sub-optimal. Important factors are the quality of the RF circuitry (design) and the antennas. Because of the hardware, the range of communications for those circuits is so short, for example for the TTGO the range is no much more than 600 meters.

-Heltec Wifi LoRa 32 [10]

Have a white PCB and come in two different versions *(the version numbers are not used by Heltec)*:

V1: with on-board PCB WiFi/Bluetooth antenna. Appears to be available for 433MHz only.

V2: with small on-board helical antenna *(has a PCB antenna on the bottom but that is not connected)*.

-TTGO LoRa32 [11]

Have a black PCB and come in several different versions:

- **LoRa** (without 32): with on-board PCB Wifi/Bluetooth antenna on top. Appears to be available for 433MHz only. Also available without the display.
- **LoRa32 V1**: with on-board metal Wifi/Bluetooth antenna on bottom. I-Pex connector located on top, it has a SX1276 LoRa transceiver module, and a OLED display controlled by a SSD1306. This will be our particular selection.
- **LoRa32 V2**: with on-board metal Wifi/Bluetooth antenna on bottom (in a different location).
  Uses ESP32-Pico-D4 (with integrated flash memory) instead of ESP32, uses a (shielded) LoRa module, I-Pex connector located on the bottom, micro-USB connector is rotated 90 degrees, in addition has a micro-SD card slot on the bottom and an on/off switch for the battery next to the micro-USB connector. Switches the battery only so not possible to switch the board off when connected to USB for charging the battery.
  'Programmable' LED on pin 22 instead of pin 25 but useless because wired to SCL and all three LEDs are on the bottom side where you cannot see them. DIO1 and DIO2 each have a separate board pin but neither of them is connected to a GPIO port so must be explicitly wired. This also means that the TTGO V2 has two GPIO ports less that could otherwise have been used for other purposes.

2.3. Environmental sensors and their data acquisition.

2.3.1. Humidity and temperature sensor

Based on a unique capacitive cell, these relative humidity sensors [12] are designed for high volume, cost sensitive applications such as office automation, automotive cabin air control, home appliances, and industrial process control systems. They are also useful in all applications where humidity compensation is needed.

One of the advantages of this sensor is that its measurements has a digital output, this while protects us more from non-desirable noises.

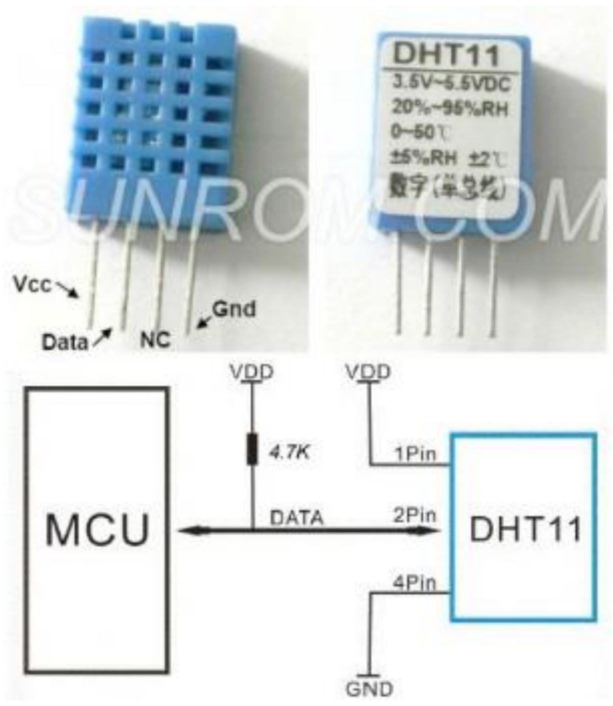
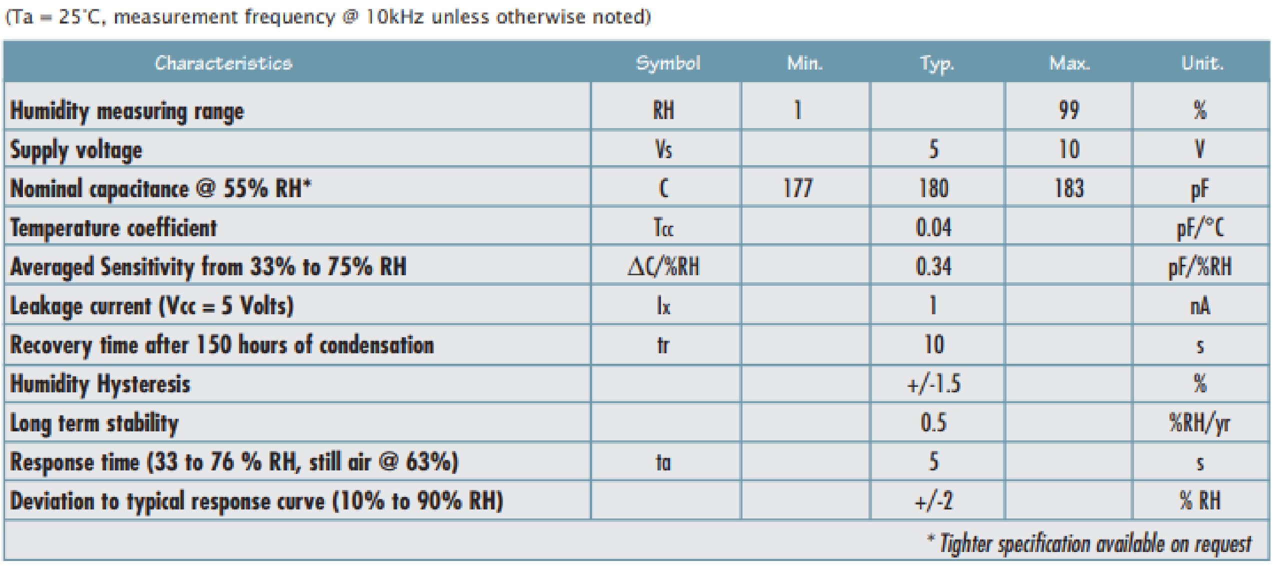


Figure 21 DHT11 with his pinout and how to connect it with a controller [12]



(Ta = 25°C, measurement frequency @ 10kHz unless otherwise noted)

Characteristics	Symbol	Min.	Typ.	Max.	Unit.
Humidity measuring range	RH	1		99	%
Supply voltage	Vs		5	10	V
Nominal capacitance @ 55% RH*	C	177	180	183	pF
Temperature coefficient	Tcc		0.04		pF/°C
Averaged Sensitivity from 33% to 75% RH	ΔC/%RH		0.34		pF/%RH
Leakage current (Vcc = 5 Volts)	Ix		1		nA
Recovery time after 150 hours of condensation	tr		10		s
Humidity Hysteresis			+/-1.5		%
Long term stability			0.5		%RH/yr
Response time (33 to 76 % RH, still air @ 63%)	ta		5		s
Deviation to typical response curve (10% to 90% RH)			+/-2		% RH

** Tighter specification available on request*

Table 1 Characteristics of the DHT11, (Ta=25º and the measurement frequency is 10KHz) [13]

2.3.2. HL 69 Soil hygrometer moisture sensor

This is an analog reading data sensor it provides us the humidity percentage of the ground [13], but it can also work as a digital sensor, will set to high when the the soil humidity exceeds a set threshold value previously selected, giving us a 1 or 0 value, but this second mode it´s not so accurate and useful.

The digital output can be connected directly to an Arduino to detect high and low by the microcontroller to detect soil moisture. A typical example of use is when we directly drive the digital outputs to the buzzer module, which can be used as a soil moisture alarm equipment. Analog output AO connected through the AD converter, you can get more precise values of soil moisture sent to your MCU.

Characteristics:

Operating voltage: 3.3V~5V
– Dual output mode – Digital or Analog – analog output more accurate
– Fixed bolt hole for easy installation
– Power indicator (red) and digital switching output indicator (green)
– LM393 comparator chip, stable
– Panel PCB Dimension: Approx.3cm x 1.5cm
– Soil Probe Dimension: Approx. 6cm x 3cm
– Cable Length: Approx.21cm

Interface Description(4-wire):
VCC: 3.3V-5V
GND: GND
DO: digital output interface (0 and 1)
AO: analog output interface

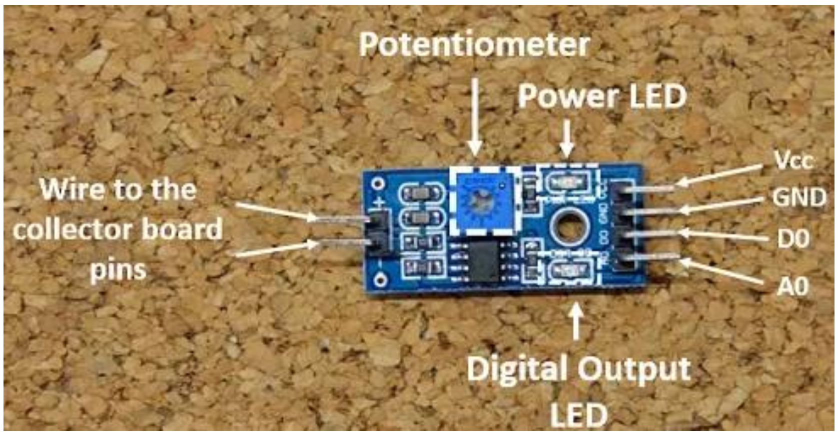


Figure 22 pinout of the controller of the HL69 [13]

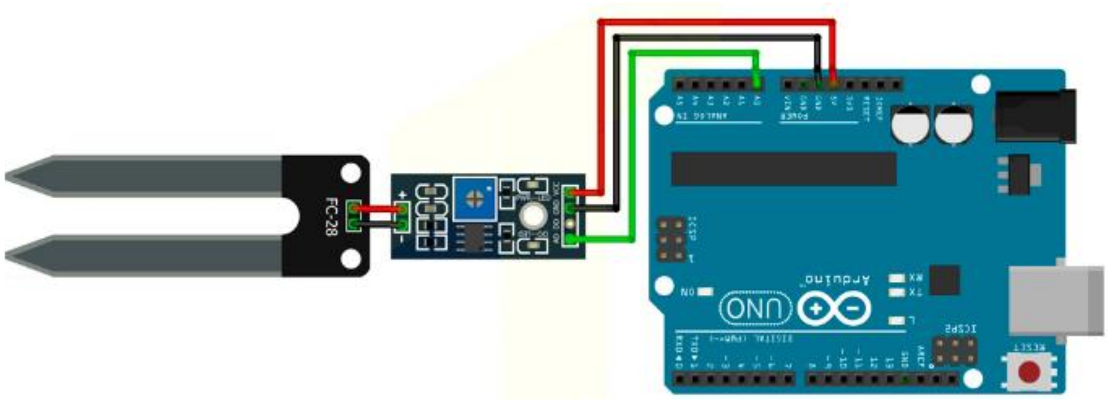


Figure 23 Arduino example of how to connect the two parts of this sensor to read data [13]

2.3.3. Ultrasonic distance sensor HC-SR04

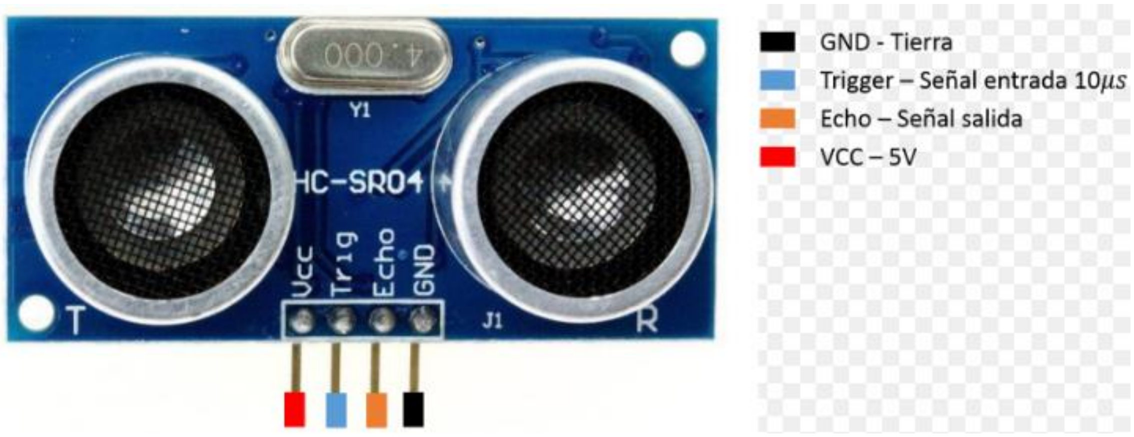


Figure 24 pinout of the HC-SR04 [14]

The HC-SR04 [14] sensor its ultrasonic distance sensor able to detect objects and calculate the distance of them to it, but this distances has to be on a range between 2 and 450 cm for the sensor to work properly.

This sensor works using ultrasonic waves and has all the electronic responsible of making the measurement and providing a trigger signal. Its way of use it´s simple, the device send a starting pulse and then measuring the length of the return pulse.

Each HC-SR04 module includes an ultrasonic transmitter, a receiver and a control circuit. It sends one pulse through the transmitter and this pulse reflects and come back to the sensor through the receiver we receive that information.

It has a low consuption and great accuracy for its reduced price.

Characteristics

- Circuit dimensions: 43 x 20 x 17 mm
- Tensión de alimentación: 5 Vcc
- Working Current 15mA
- Measuring Angle 15 degree
- Frecuencia de trabajo: 40 KHz
- Maximum range: 4.5 m
- Minimum range: 1.7 cm
- Minimum trigger pulse duration (TTL level Time-to-live): 10 µS.
- Eco output pulse duration(TTL level Time-to-live): 100-25000 µS.
- Minimum waiting time between one measure and the start of the next one: 20 mS.

Pinout:

- VCC
- Trig (*Ultrasoud launch*)
- Echo (*ultrasound reception*)
- GND

We could easily calculate the distance to the target with this simple ecuation:

*Distance = {(Between trig and echo time) * (speed of sound 340 m/s)}/2*

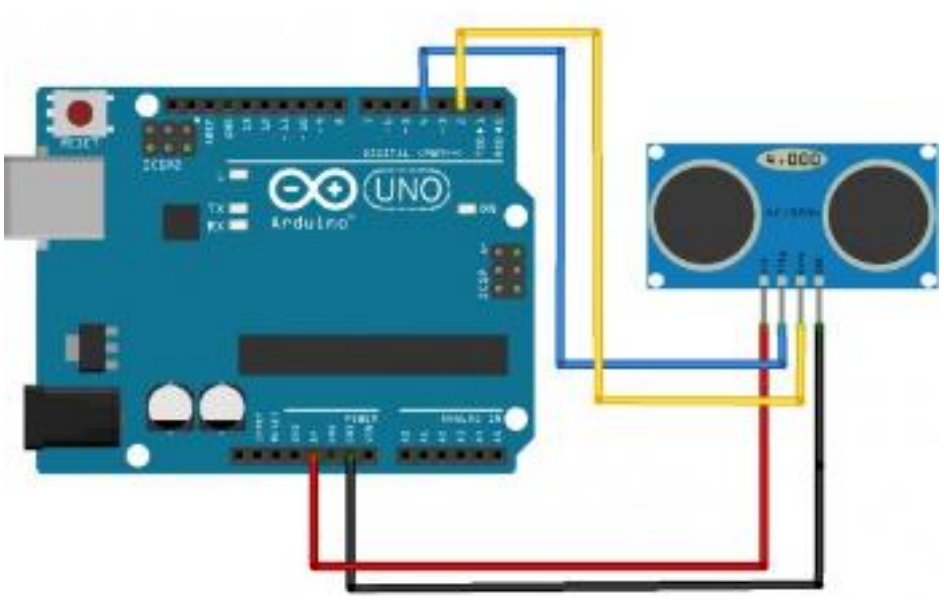


Figure 25 example of how to read data from the HC-SR04 with an Arduino UNO [14]

From a more technical perspective the pulse transmission will be more explained in detail: You only need to supply a short 10uS pulse to the trigger input to start the ranging, and then the module will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo.

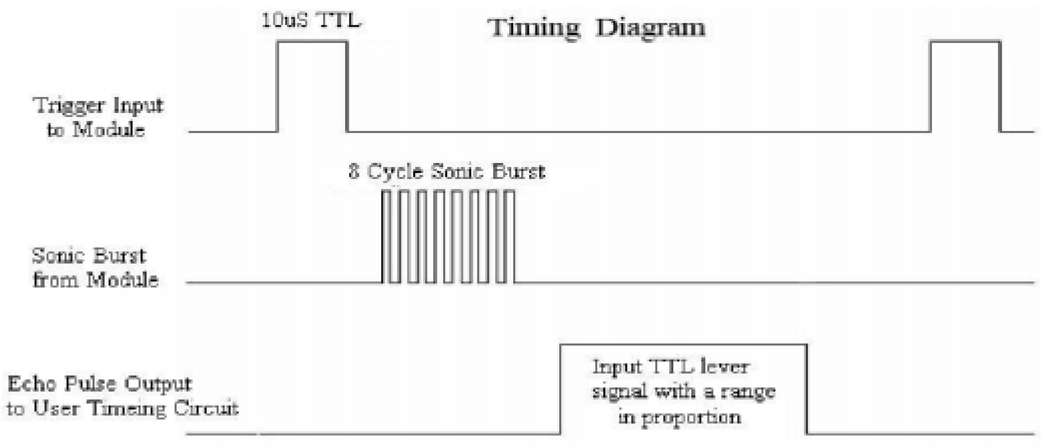

10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion

Figure 26 HC-SR04 trigger and echo transmission timing diagram [14]

2.3.4. Sensor de temperature LM35

This is going to be our selection, so we will talk about it later.

2.3.5. MMA7361 Accelerometer sensor

Figure 27 MMA736 accelerometer sensor [15]

This sensor is an analog accelerometer of 3 exes (x,y,z) [15]. The level of the measurements of the accelerometer allows us to measure the acceleration or the inclination of a platform or surface of a object in regards to the earth´s surface.

Another features of this sensor are: the sleep mode, the signal conditioning, low pass filter of 1 pole, temperature compensation, auto-test and 0 g detection for free fall. This sensor works with power sources between 2.2 V and 3.6 VDC (3.3 V is the optimal value).

Sensor specifications MMA7361:

- Low power current: 400 µA.
- Waiting mode: 3 µA.
- Low operating voltage: 2,2 V – 3,6 V.
- High sensibility (800 mV / g@1.5g).
- Selectable sensibility (± 1,5 g, ± 6 g).
- 0g- Detect for the free fall protection.
- signal condiotioning with the low pass filter.

Sensor pinout:

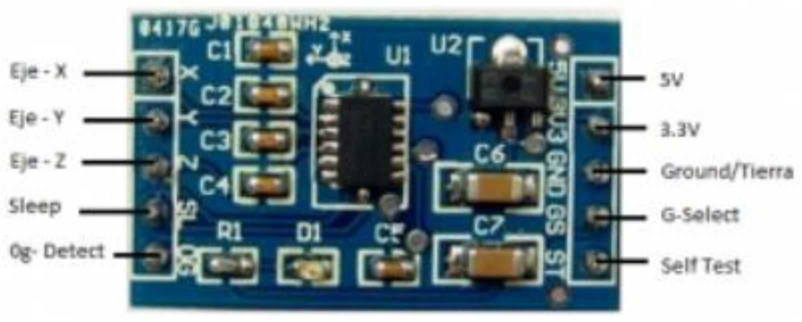


Figure 28 MMA736 pinout [15]

Pinout of the MMA736

- **X exe:** It is an analog output signal along the X axis.
- **Y exe:** It is an analog output signal along the Y axis.
- **Z exe:** It is an analog output signal along the Z axis.
- **Sleep(SL):** This pin is activated in a negated way the integrated one will go to sleep and will not send us anything in its outputs. operation will resume when more power is consumed.
- **Detect (0G):** This pin will be high when 0 g is detected on all 3 axes. Useful for detecting free fall
- **5 V:** This pin is connected to a built in regulator that will bring 5v to 3.3v required on the chip to run
- **3.3 V:** This pin does not pass through the 5v regulator, it is for those who have a previously regulated 3.3v voltage.
- **Ground (GND):** This pin must be connected to circuit ground.
- **Sense select (GS):** This pin is an enabling pin for the x, y, and z sensors. This must be enabled at low level, and then the power and ground pins are subtracted. is used to select between the two sensitivities. If this pin is low, it is in 1.5 g mode. If it is high, it switches to 6 g mode.
- **Self Test (ST):** This chip has been built in a self-test to verify that both the mechanical and electrical parts inside the chip are working properly.

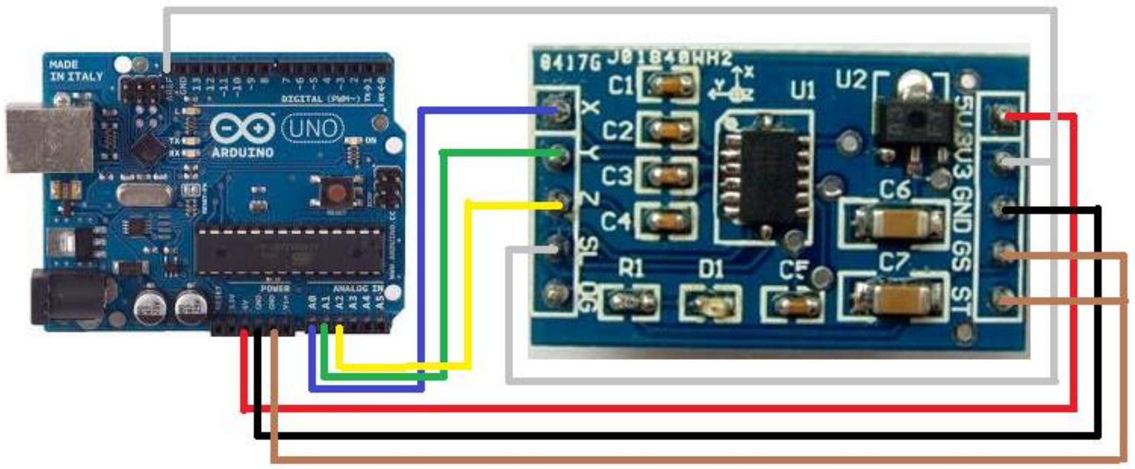


Figure 29 Arduino pin connection example [15]

3.DESIGN PART.

3.1. Sensor selection and requirements for data reading.

The LM35 it’s an analog temperature sensor and differs with the classic thermistors in the way of measure the degrees, the classical thermistors measure it changing its electric resistance. The LM35 it’s an integrated with its own control circuit, which provides an analog output voltage in proportion to the temperature.

The output of the LM35 is lineal with the temperature, increasing it 10 mV for each degree centigrade. The acceptable measure range goes from the -55ºC (-550mV) to the 150ºC (150mV). Its accuracy at ambient temperature is 0.5ºC (25ºC). [16]

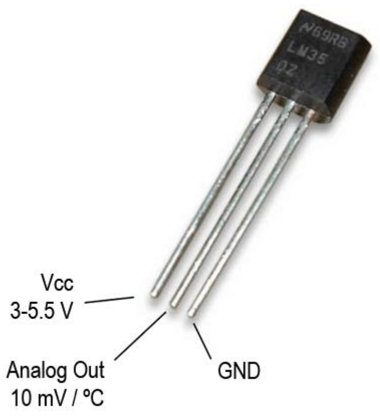


Figure 30 Pinout of the LM35 [16]

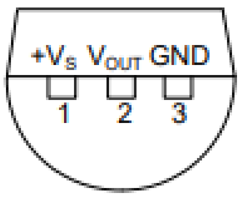


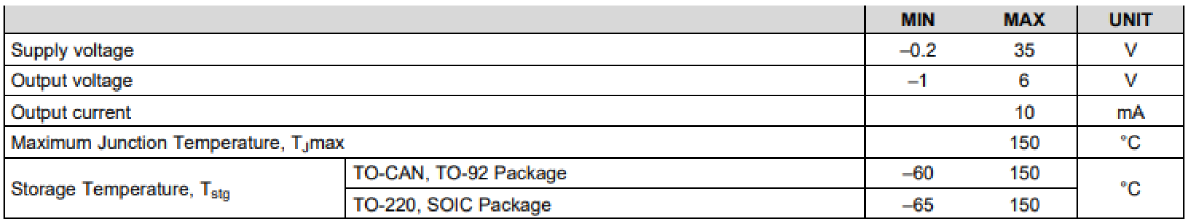
Figure 31 Bottom view of LM35 LP package [16]

For our own purpose case we have chosen a simple sensor, in order to not take risks in the data reading process. This detects temperatures from -55ºC to 150ºC, 1ºC is equivalent to 10mV and supports voltages between 4V and 30V. All this information has been obtained from the LM35 data sheet.

Technical features:

- Linear + 10-mV/°C Scale Factor
- 0.5°C Ensured Accuracy (at 25°C)
- Rated for Full −55°C to 150°C Range
- Suitable for Remote Applications
- Low-Cost Due to Wafer-Level Trimming
- Operates From 4 V to 30 V
- Less Than 60-µA Current Drain
- Low Self-Heating, 0.08°C in Still Air
- Non-Linearity Only ±¼°C Typical
- Low-Impedance Output, 0.1 Ω for 1-mA Load
- Calibrated directly in Celsius (Centigrade)

Maximum supported electrical ratings:



		MIN	MAX	UNIT
Supply voltage		–0.2	35	V
Output voltage		–1	6	V
Output current			10	mA
Maximum Junction Temperature, $T_J$max			150	°C
Storage Temperature, T_{stg}	TO-CAN, TO-92 Package	–60	150	°C
	TO-220, SOIC Package	–65	150	

Table 2 Maximum electrical ratings for the LM35 [16]

It’s also recommended for this sensor to work between -55ºC and 150ºC.

Application form:

The manufacturer gives a bunch of different possibilities to implement our little temperature measurement station, starting with the simplest one which has a shortened range of measure of 2º to 150º; to a lot of more complex implementation, like the one of the figure 33, where we have the full range of measure from -55º to 150º.

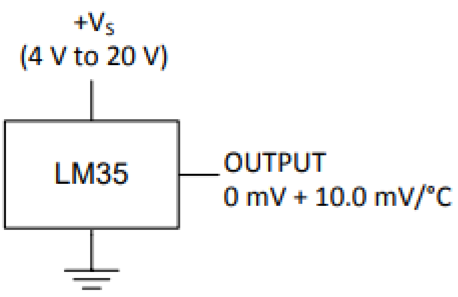
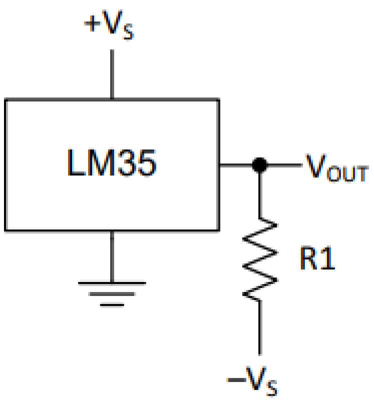


Figure 32 basic temperature sensor (2ºC to 150ºC) [16]



Choose $R_1 = -V_S / 50\ \mu A$
V_{OUT} = 1500 mV at 150°C
V_{OUT} = 250 mV at 25°C
V_{OUT} = –550 mV at –55°C

Figure 33 full range centigrade temperature sensor [16]

We have selected the first type of implementation, because we don´t need a great range of measurements for this project, we can manage with the easy one, moreover at this time of the year there is no possibility to record values below 2º. So our range of sensing will be between 2ºC and 100ºC more or less, it will be powered with 5 V while the ADC of the ESP 32 can just read between 0 V and 3.3 V, but as we have said, with the values that we are going to work it won´t be a problem, we won´t break the board.

Possible error on the measure:

We look at the LM35 line and we can see that the optimal working temperature is the 25º, but if we increment or decrement the ambient temperature we start adding a linear growing error, that increase from ± 1º to ± 1.5º. [17]

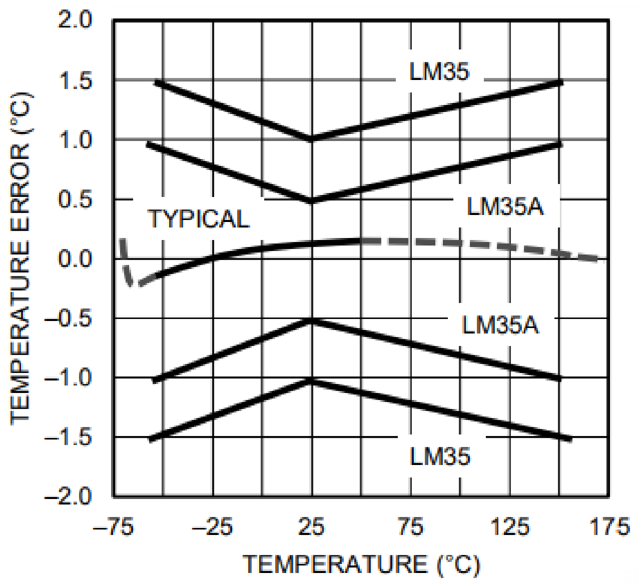


Figure 34 Accuracy vs Temperature (Ensured) [16]

Power supply advice:

This sensor has wide range of valid values (4V to 30V), so it's an easy to use-and-success IC for many applications. But for noisy environments is recommended to add a 0.1 uF capacitor from the V+ to the GND to bypass the supply voltage and make it more stable, reducing the total amount of noise.

Linearity of the ESP32 ADC measures with the LM35:

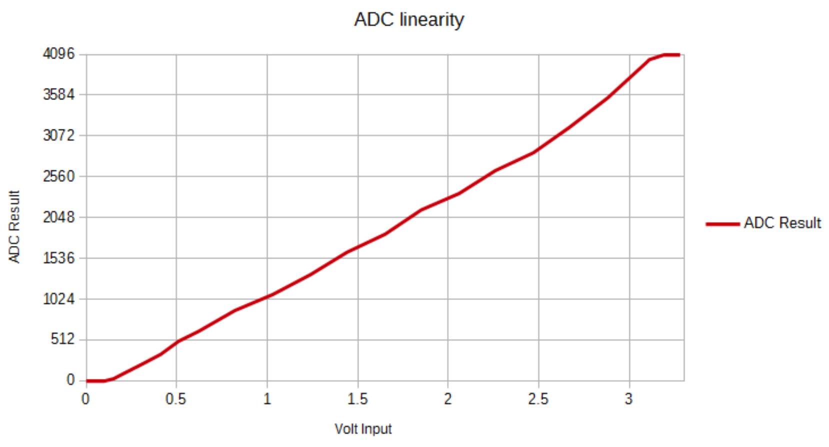


Figure 35 sensor volt input versus the ESP32 adc output. [17]

Resolution 3.3V/4096 = 0.000805664 V = 0.8 mV

3.2. Functional diagram design.

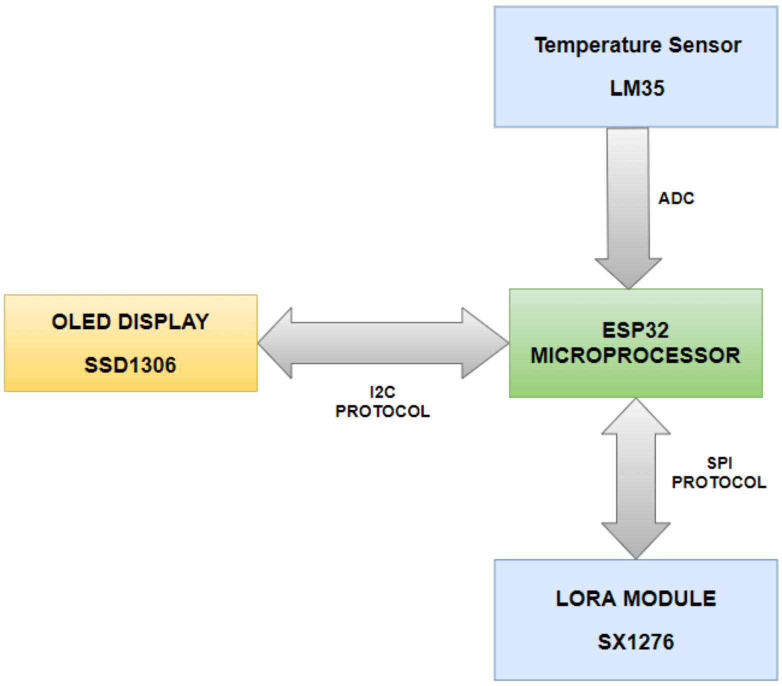


Figure 36 Simple Connections Diagram

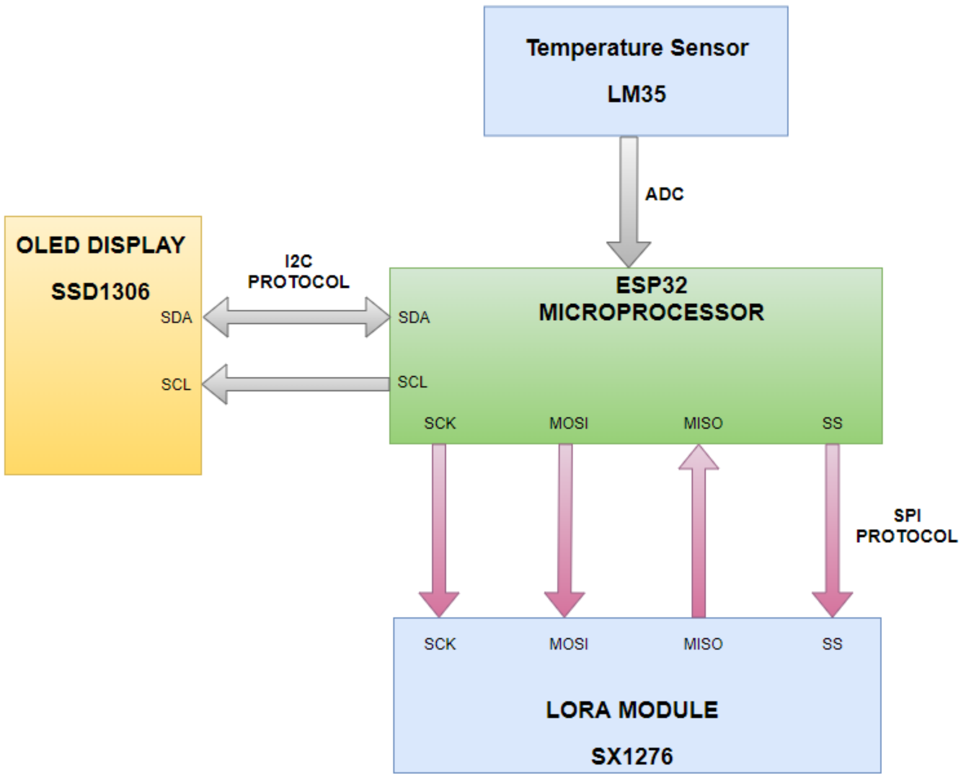


Figure 37 Complex Connections Diagram

3.3. Microcontroller selection. Parameters, connectivity and toolset of ESP32.

3.3.1. Introduction

Created by Espressif Systems, ESP32 is a low-cost, low-power system on a chip (SoC) series with Wi-Fi & dual-mode Bluetooth capabilities.

At its heart, there's a dual-core or single-core Tensilica Xtensa LX6 microprocessor with a clock rate of up to 240 MHz. ESP32 is highly integrated with built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power management modules. Engineered for mobile devices, wearable electronics, and IoT applications, ESP32 achieves ultra-low power consumption through power saving features including fine resolution clock gating, multiple power modes, and dynamic power scaling. [18]

3.3.2. Parameters of ESP32

- Processors:
  - Main processor: Tensilica Xtensa 32-bit LX6 microprocessor
    - Cores: 2 or 1 (depending on variation) All chips in the ESP32 series are dual-core except for ESP32-S0WD, which is single-core.
    - Clock frequency: up to 240 MHz
    - Performance: up to 600 DMIPS
  - Ultra-low power co-processor: allows you to ADC conversions, computation, and level thresholds while in deep sleep.
- Wireless connectivity:
  - Wi-Fi: 802.11 b/g/n/e/i (802.11n @ 2.4 GHz up to 150 Mbit/s)
  - Bluetooth: v4.2 BR/EDR and Bluetooth Low Energy (BLE)
- Memory:
  - Internal memory:
    - ROM: 448 KiB For booting and core functions.
    - SRAM: 520 KiB For data and instruction.

        - RTC fast SRAM: 8 KiB For data storage and main CPU during RTC Boot from the deep-sleep mode.
        - RTC slow SRAM: 8 KiB For co-processor accessing during deep-sleep mode.
        - eFuse: 1 Kibit Of which 256 bits are used for the system (MAC address and chip configuration) and the remaining 768 bits are reserved for customer applications, including Flash-Encryption and Chip-ID.
        - Embedded flash: Flash connected internally via IO16, IO17, SD_CMD, SD_CLK, SD_DATA_0 and SD_DATA_1 on ESP32-D2WD and ESP32-PICO-D4.
            - 0 MiB (ESP32-D0WDQ6, ESP32-D0WD, and ESP32-S0WD chips)
            - 2 MiB (ESP32-D2WD chip)
            - 4 MiB (ESP32-PICO-D4 SiP module)
    - External flash & SRAM: ESP32 supports up to four 16 MiB external QSPI flashes and SRAMs with hardware encryption based on AES to protect developers' programs and data. ESP32 can access the external QSPI flash and SRAM through high-speed caches.
        - Up to 16 MiB of external flash are memory-mapped onto the CPU code space, supporting 8-bit, 16-bit and 32-bit access. Code execution is supported.
        - Up to 8 MiB of external flash/SRAM memory are mapped onto the CPU data space, supporting 8-bit, 16-bit and 32-bit access. Data-read is supported on the flash and SRAM. Data-write is supported on the SRAM. ESP32 chips with embedded flash do not support the address mapping between external flash and peripherals.
- Peripheral input/output: Rich peripheral interface with DMA that includes capacitive touch, ADCs (analog-to-digital converter), DACs (digital-to-analog converter), I²C (Inter-Integrated Circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (Controller Area Network), SPI (Serial Peripheral Interface), I²S (Integrated Inter-IC Sound), RMII (Reduced Media-Independent Interface), PWM (pulse width modulation), and more.

- Security:
    - IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
    - Secure boot
    - Flash encryption
    - 1024-bit OTP, up to 768-bit for customers
    - Cryptographic hardware acceleration: AES, SHA-2, RSA, elliptic curve cryptography (ECC), random number generator (RNG)

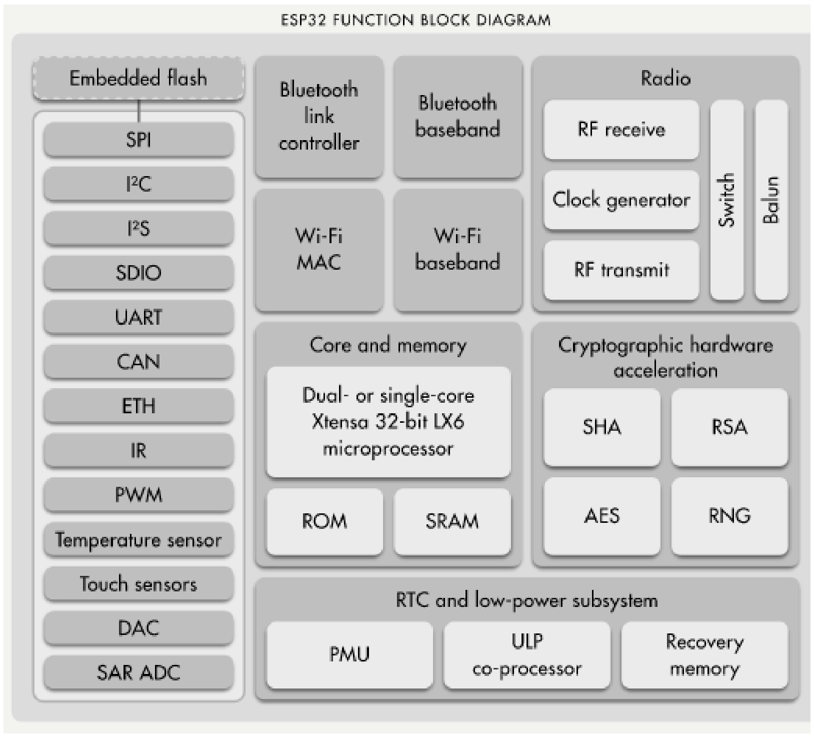


Figure 38 ESP 32 block diagram [18]

3.3.3. Toolset

Steps to install Arduino ESP32 support on Windows [19]:

*Tested on 32 and 64 bit Windows 10 machines

1. Download and install the latest Arduino IDE Windows Installer from arduino.cc

2. Download and install Git from git-scm.com

3. Start Git GUI and run through the following steps:

- Select Clone Existing Repository

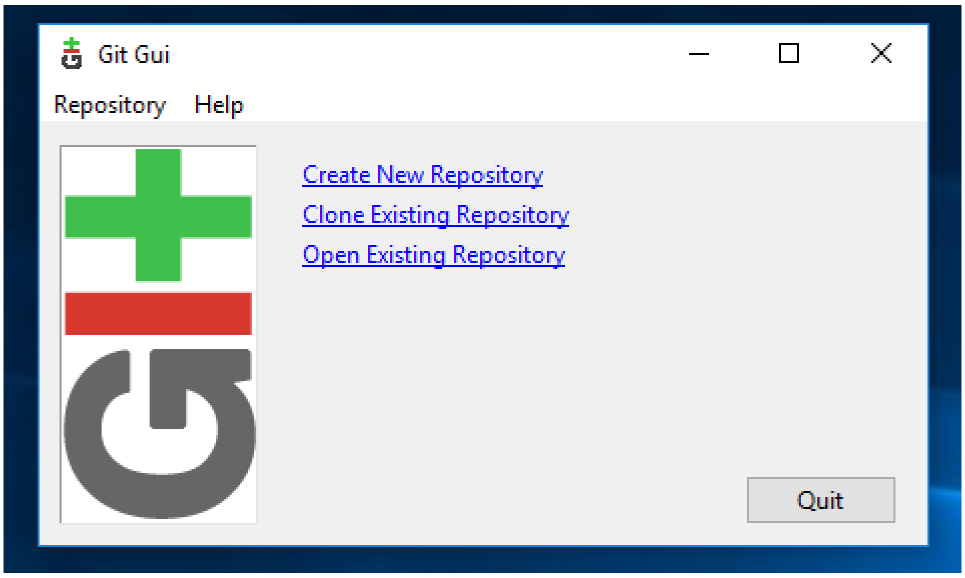
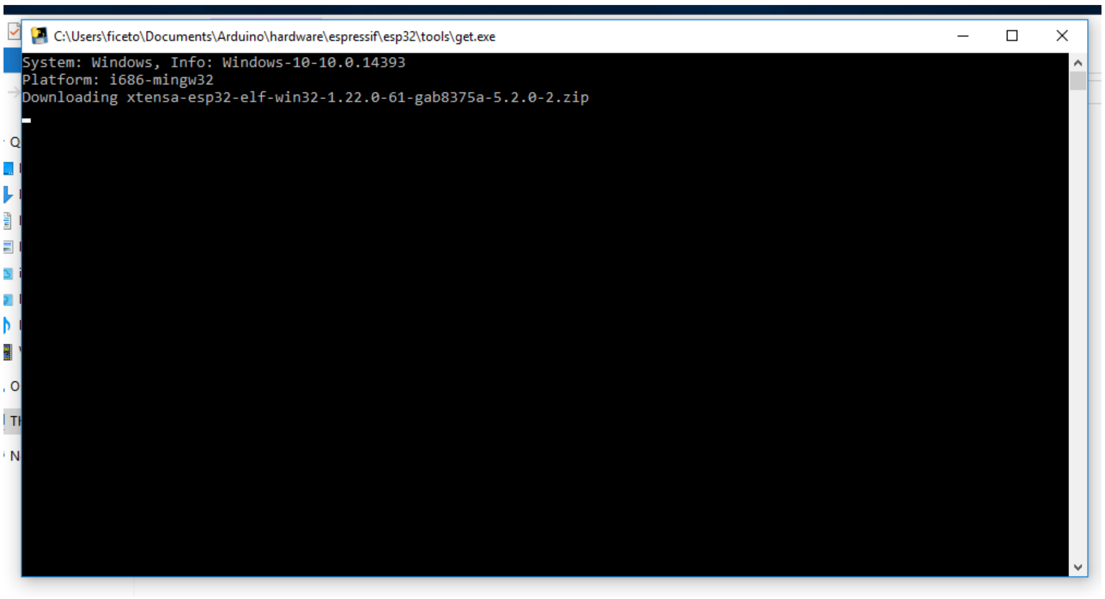


Figure 39 Git Gui menu

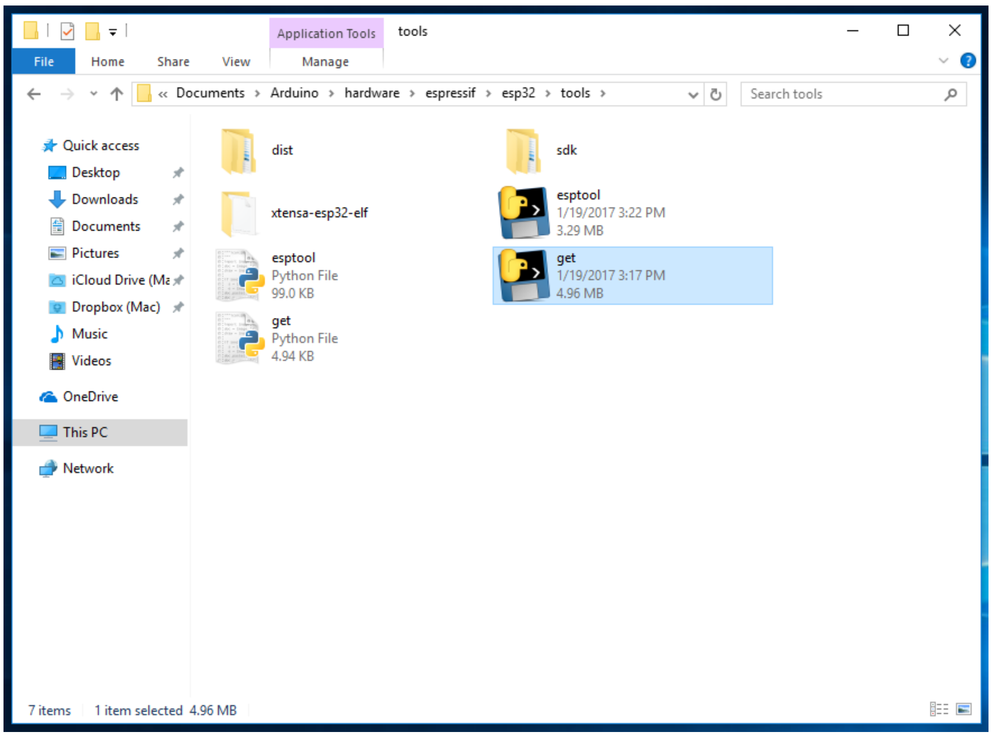
- Select source and destination
    - Sketchbook Directory: Usually C:/Users/[YOUR_USER_NAME]/Documents/Arduino and is listed underneath the "Sketchbook location" in Arduino preferences.
    - Source Location: https://github.com/espressif/arduino-esp32.git
    - Target Directory: [ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32
    - Click Clone to start cloning the repository
- open a Git Bash session pointing to [ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32 and execute git submodule update --init --recursive
- Open [ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32/tools and double-click get.exe



```
C:\Users\ficeto\Documents\Arduino\hardware\espressif\esp32\tools\get.exe
System: Windows, Info: Windows-10-10.0.14393
Platform: i686-mingw32
Downloading xtensa-esp32-elf-win32-1.22.0-61-gab8375a-5.2.0-2.zip
```

Figure 40: execution of get.exe

- When get.exe finishes, you should see the following files in the directory



```
« Documents › Arduino › hardware › espressif › esp32 › tools ›

dist                          sdk
xtensa-esp32-elf              esptool
                              1/19/2017 3:22 PM
                              3.29 MB
esptool                       get
Python File                   1/19/2017 3:17 PM
99.0 KB                       4.96 MB
get
Python File
4.94 KB

7 items    1 item selected  4.96 MB
```

Figure 41 new get files

4. Plug your ESP32 board and wait for the drivers to install (or install manually any that might be required)

5. Start Arduino IDE
6. Select your board in Tools > Board menu
7. Select the COM port that the board is attached to
8. Compile and upload (You might need to hold the boot button while uploading)

ESP 32 Pinout:

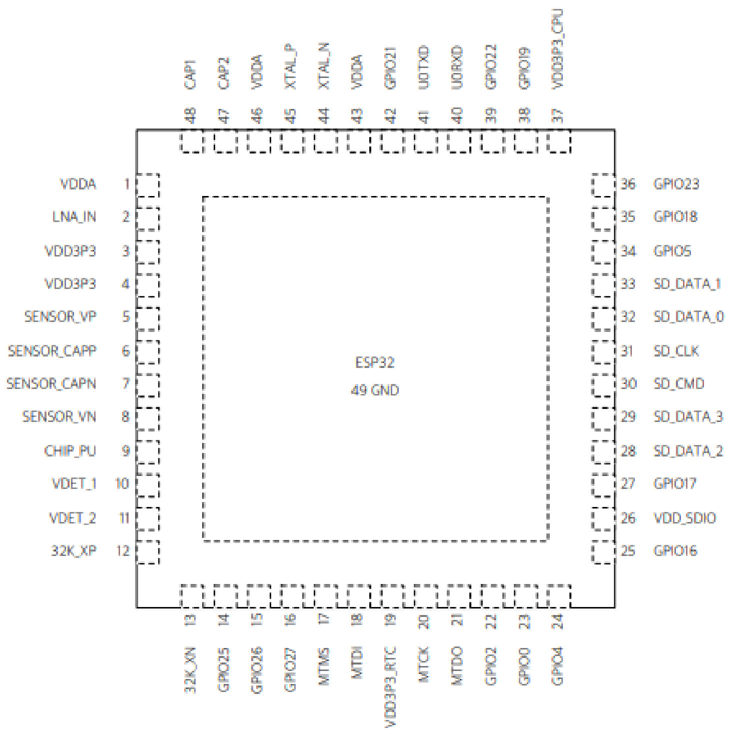


Figure 42 ESP 32 pinout [20]

As the pinout of the ESP 32 is closely related to the TTGO Lora module V1 I put the TTGO pinout here, because a lot of pins come directly from ESP32.

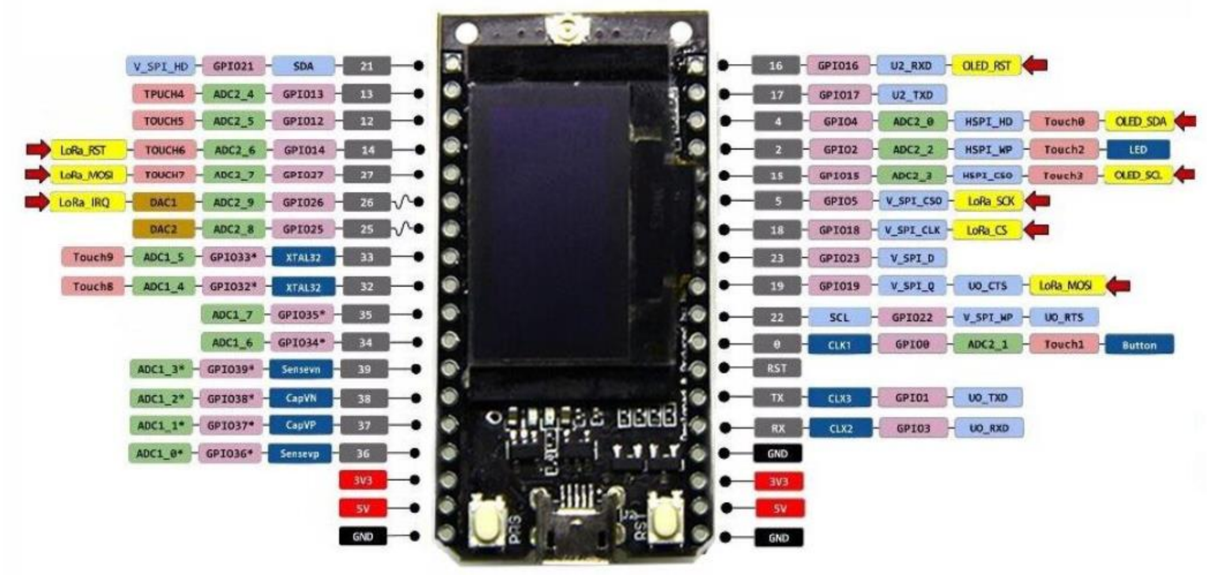


Figure 43 TTGO ESP32 LoRa module V1 [19]

3.4. Lora chip selection. Parameters, connectivity and programming of SX1276.

3.4.1. Introduction

Semtech’s LoRa transceivers feature a long-range wireless modem that provides ultra-long range spread spectrum communication and high interference immunity while minimizing current consumption.

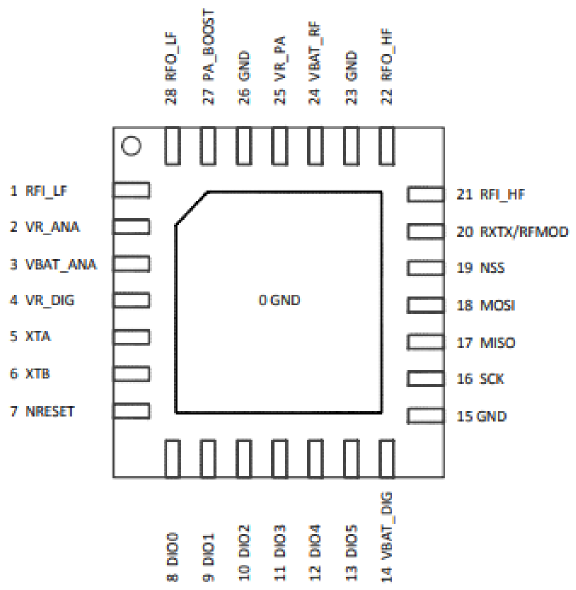


Figure 44 SX1276 pinout [21]

3.4.2. Parameters

- integrated +20dBm power amplifier LoRa Modem
- 168dB maximum link budget
- +20dBm - 100 mW constant RF output vs. V supply
- +14dBm high efficiency Power Amplifier
- Programmable bit rate up to 300kbps
- High sensitivity: down to -148dBm
- Bullet-proof front end: IIP3 = -11dBm
- Excellent blocking immunity
- Low RX current of 9.9mA, 200nA register retention
- Fully integrated synthesizer with a resolution of 61Hz
- FSK, GFSK, MSK, GMSK, LoRa and OOK modulation
- Built-in bit synchronizer for clock recovery
- Preamble detection
- 127dB Dynamic Range RSSI
- Automatic RF Sense and CAD with ultra-fast AFC
- Packet engine up to 256 bytes with CRC
- Built-in temperature sensor and low battery indicator

3.4.3. Connectivity

To connect LoRa module with the ESP32 microprocessor we are using the SPI protocol with Arduino, so a brief introduction to his protocol will be needed:

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two microcontrollers.

With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices. Typically, there are three lines common to all the devices:

- MISO (Master In Slave Out) - The Slave line for sending data to the master,
- MOSI (Master Out Slave In) - The Master line for sending data to the peripherals,
- SCK (Serial Clock) - The clock pulses which synchronize data transmission generated by the master

And one line specific for every device:

- SS (Slave Select) - the pin on each device that the master can use to enable and disable specific devices.

When a device's Slave Select pin is low, it communicates with the master. When it's high, it ignores the master. This allows you to have multiple SPI devices sharing the same MISO, MOSI, and CLK lines

Here it is show and example of the SPI protocol working with a time diagram. This interface gives access to a configuration register to select and configure how are is the IC going to access the register, we have: single, burst and FIFO. They difference each other in the way that the frame is composed, in the first one after the address byte, there is only one data byte, in the second there are a bunch of them like in the third, so on the last two you don´t need the address byte to be sent each time, it´s enough with the first address byte, then the BURST method increment its address value and the FIFO the address is memorized [21].

SINGLE access: an address byte followed by a data byte is sent for a write access whereas an address byte is sent, and a read byte is received for the read access.

The NSS pin goes low at the beginning of the frame and goes high after the data byte.

The figure below shows a typical SPI single access to a register.

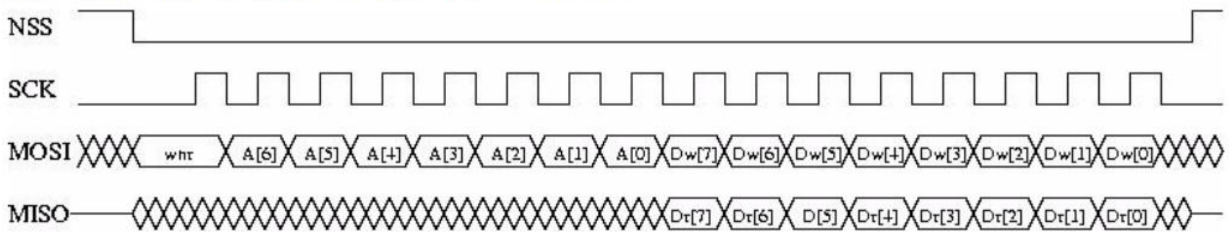


Figure 45 SPI timing diagram (SINGLE access) [21]

MOSI is generated by the master on the falling edge of SCK and is sampled by the slave (i.e. this SPI interface) on the rising edge of SCK. MISO is generated by the slave on the falling edge of SCK. A transfer is always started by the NSS pin going low. MISO is high impedance when NSS is high.

The first byte is the address byte. It is comprising:

- A wnr bit, which is 1 for write access and 0 for read access.
- Then 7 bits of address, MSB first. The second byte is a data byte, either sent on MOSI by the master in case of a write access or received by the master on MISO in case of read access. The data byte is transmitted MSB first.

Proceeding bytes may be sent on MOSI (for write access) or received on MISO (for read access) without a rising NSS edge and re-sending the address. In FIFO mode, if the address was the FIFO address then the bytes will be written / read at the FIFO address. In Burst mode, if the address was not the FIFO address, then it is automatically incremented for each new byte received. The frame ends when NSS goes high. The next frame must start with an address byte. The SINGLE access mode is therefore a special case of FIFO / BURST mode with only 1 data byte transferred. During the write access, the byte transferred from the slave to the master on the MISO line is the value of the written register before the write operation [22].

As we are going to use a SPI protocol to communicate the SX1276 LoRa module with the ESP32, the pin mapping will be according to the OLED_LoRa_sender.ino code (first board test), (Remember we are using V1):

Definition name in Arduino	Pin mapping	Type of the pin in TTGO Lora Module	
SCK	Pin 5	GPIO 5	LoRa_SCK
MISO	Pin 19	GPIO 19	LoRa_MOSI
MOSI	Pin 27	GPIO 27	LoRa_MOSI
SS	Pin 18	GPIO 18	LoRa_CS
RST	Pin 14	GPIO 14	LoRa_RST
DIO	Pin 26	GPIO 26	LoRa_IRQ

Table 3 pin connection SX1276 with ESP32

3.4.4. Programming

The first thing we need to start writing code to control the SX1276 transceiver is to include a couple of useful and must-install Arduino libraries:

The first one the Lora.h library, for sending and receiving data using LoRa radios, supports Semtech SX1276 base boards and shields.

The SPI.h library, this library allows you to communicate with SPI devices, with the Arduino as the master device.

The Wire.h library, this library allows you to communicate with I2C / TWI devices.

The SSD1306.h is one option of methods to talk with the display through the controller, it has drawing functions, string printing functions and so on, we can see it in detail in the annexes of code.

3.5. OLED display with SSD1306 controller connection and programming.

3.5.1. Introduction

An OLED display its similar to a LED/LCD display, in which the light is provided by a external source, besides the OLED display use organic LEDs that has autonomous lightning, this feature makes the OLED displays even thinner, and probably the thinnest technology.

The brain that will tell our display what to do is going to be the SSD1306 controller. It's a single-chip CMOS OLED/PLED driver with controller for organic/polymer light emitting diode dot-matrix graphic system. It consists of 128 segments and 64 commons. This integrated circuit is designed for Common Cathode type OLED panel.

This controller embeds with contrast control, display RAM and oscillator, which reduces the number of external components and the power consumption. It has a 256-step brightness control. Data/Commands are sent from general microcontroller unit through the hardware selectable 6800/8000 series compatible Parallel Interface, I2C interface or Serial Peripheral Interface. It is suitable for many compact portable applications, such as mobile phone sub-display, and many other applications.

3.5.2. Connection

Figure 46 SSD1306 controller and OLED display

In this picture we can see how the interface between the OLED display and his controller will take place through the common and the segment pins, which will tell which pixels have to turn up.

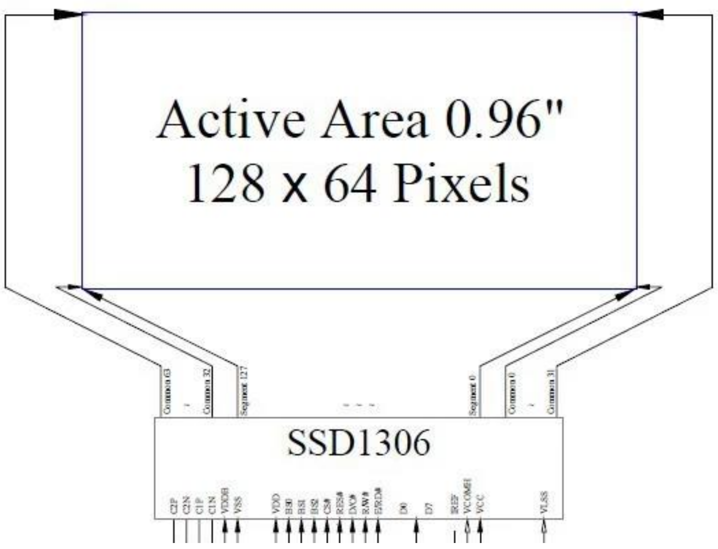


Figure 47 SSD1306 controller and OLED display (technical) [23]

In the following picture we can see the exact connections that we are supposed to do if we want to connect the SSD1306 controller with any other MCU that provides him the data to display, there also a couple of capacitors that are recommend to use, and last but not least the power source with a couple of parallel capacitors to maintain the value constant.

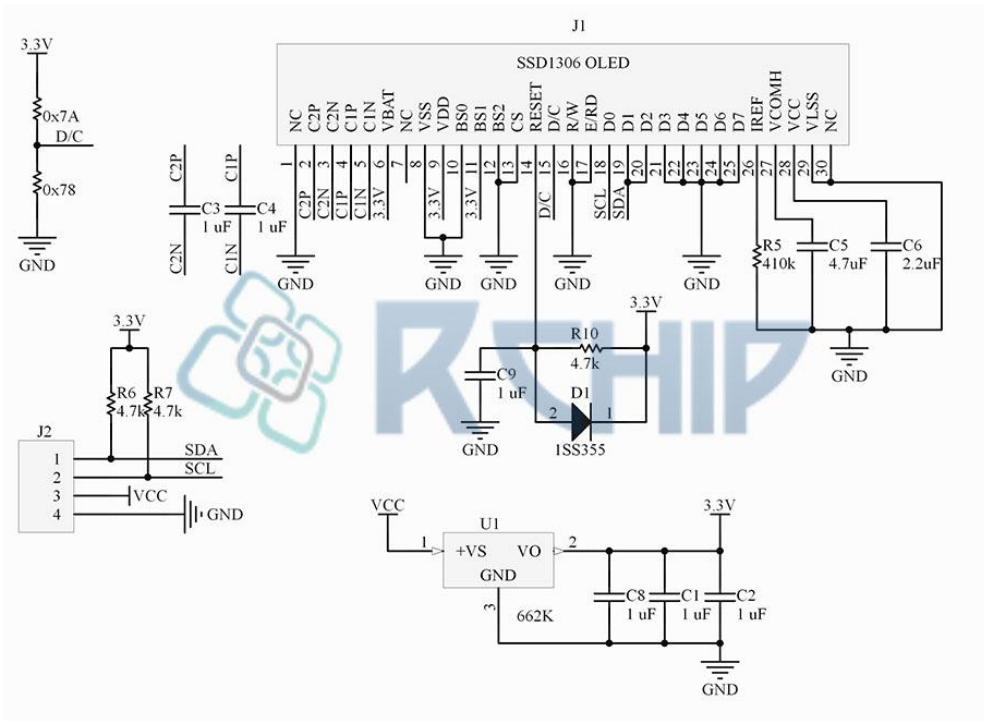


Figure 48 SSD1306 controller connections needed to work with any other device

In our controller we can difference a few determinant parts for our control of the oled display. The first one is the Graphic Display Data RAM, and the second one are the Common and Segments drivers; we will explain them later. For now we can plced them in a figurative way as we see in the next picture. [23]

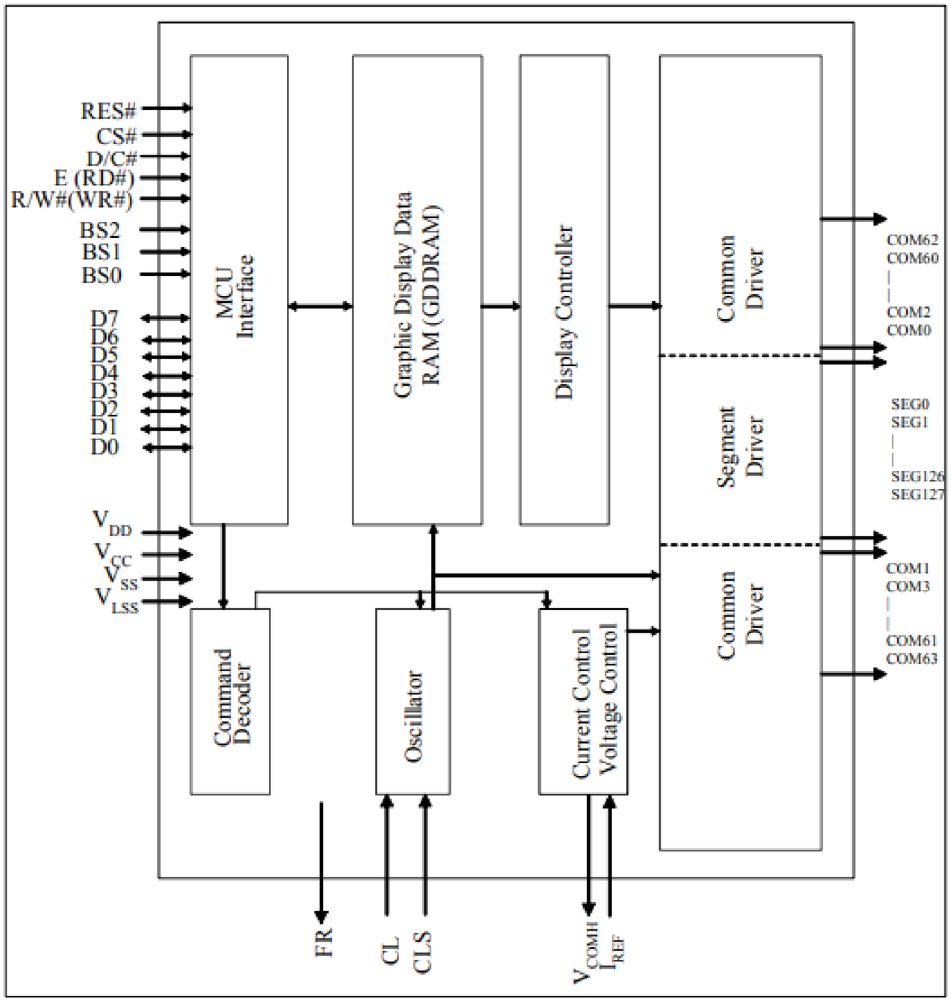


Figure 49 Block diagram of the SSD1306 OLED dislpay controller [23]

Segments driver / Common driver:

Comunication through SSD1306 and OLED its made through a segmet/common driver.

Segment drivers deliver 128 current sources to drive the OLED panel. The driving current can be adjusted from 0 to 100uA with 256 steps. Common drivers generate voltage-scanning pulses.

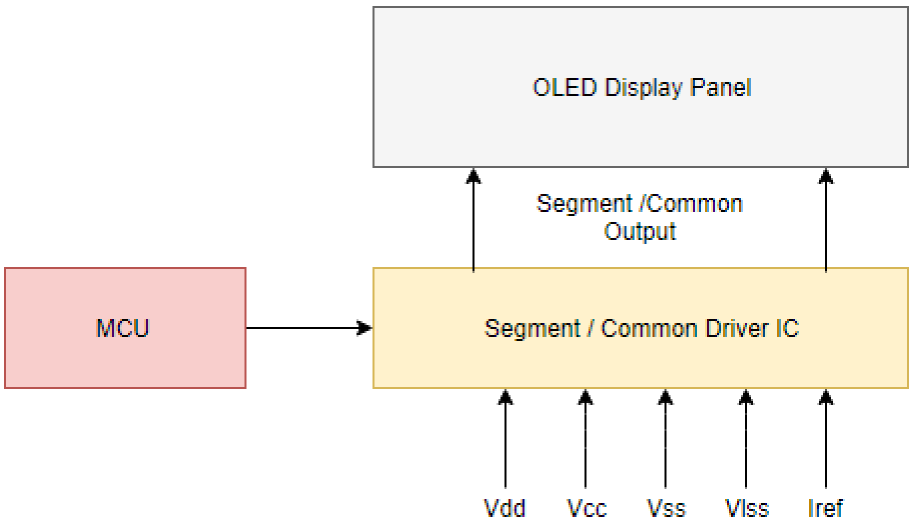


Figure 50 Communication diagram between a MCU the Seg/Com driver and the OLED display

Graphic display Data RAM (GDDRAM):

This is a bit mapped static RAM that has the pattern about to be displayed, this RAM has 128 x 64 bits and divided that ones into pages, from 0 to 7. Each page it's like a row of a matrix, containing all the segments from 0 to 127, and a group of commons, actually 8 commons each page.

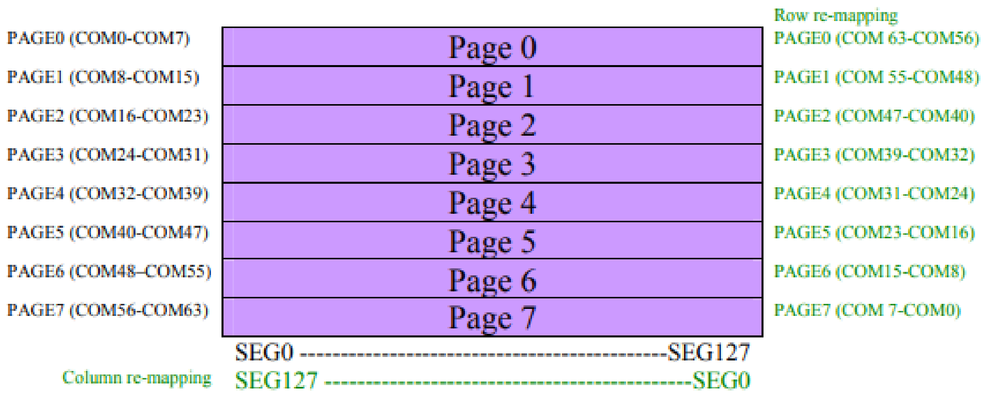


Figure 51 GDDRAM pages structure of SSD1306 [23]

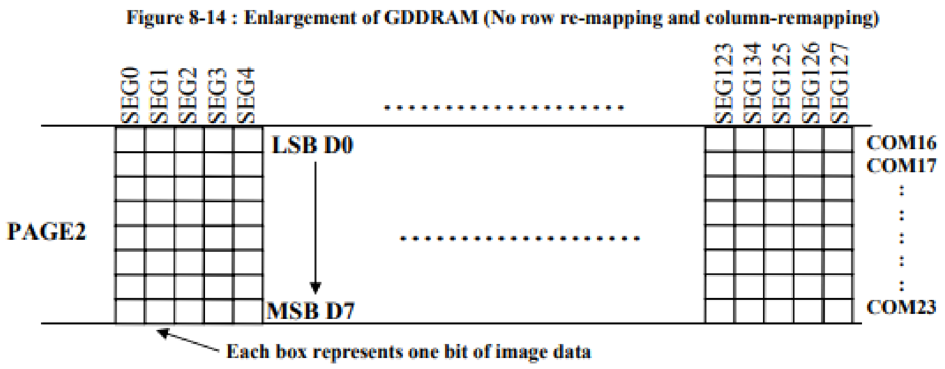


Figure 8-14 : Enlargement of GDDRAM (No row re-mapping and column-remapping)
SEG0
SEG1
SEG2
SEG3
SEG4
SEG123
SEG134
SEG125
SEG126
SEG127
LSB D0
MSB D7
PAGE2
COM16
COM17
COM23
Each box represents one bit of image data

Figure 52 Detailed pages of the SSD1306 [23]

The controller gives us the possibility to select the interface protocol that we will use to "talk" with it. This is possible because of a group of three-bit pins: BS0, BS1, BS2 that taking HIGH or Low different combinations of values will select the protocol. For example in our case, I2C communication ks selected with the combination: [BS0, BS1 , BS2] = [0 , 1 , 0]

SSD1306 Pin Name	I^2C Interface	6800-parallel interface (8 bit)	8080-parallel interface (8 bit)	4-wire Serial interface	3-wire Serial interface
BS0	0	0	0	0	1
BS1	1	0	1	0	0
BS2	0	1	1	0	0

Table 4 MCU bus interface pin selection [23]

Note:

(1) 0 is connected to VSS

(2) 1 is connected to VDD

Pin Name / Bus Interface	Data/Command Interface								Control Signal				
	D7	D6	D5	D4	D3	D2	D1	D0	E	R/W#	CS#	D/C#	RES#
8-bit 8080	D[7:0]								RD#	WR#	CS#	D/C#	RES#
8-bit 6800	D[7:0]								E	R/W#	CS#	D/C#	RES#
3-wire SPI	Tie LOW					NC	SDIN	SCLK	Tie LOW		CS#	Tie LOW	RES#
4-wire SPI	Tie LOW					NC	SDIN	SCLK	Tie LOW		CS#	D/C#	RES#
I^2C	Tie LOW					SDA_{OUT}	SDA_{IN}	SCL	Tie LOW			SA0	RES#

Table 5 MCU interface assignment under different bus interface mode [23]

As we are using the I2C, the last row will be the one. Using three digital pins to the data and commands transmission: the D2 for the SDAout, the D1 SDAin and finally the D0 for the SCL (we will explain these concepts later); mine while the other digital pins (from D7 to D3) will remain at a low level. In addition, we will need a couple of control signals: in the pin D/C# it will take place the SA0, and in the RES# pin we will have the RES# (regular reset); and the E, R/W# and CS# will remain LOW.

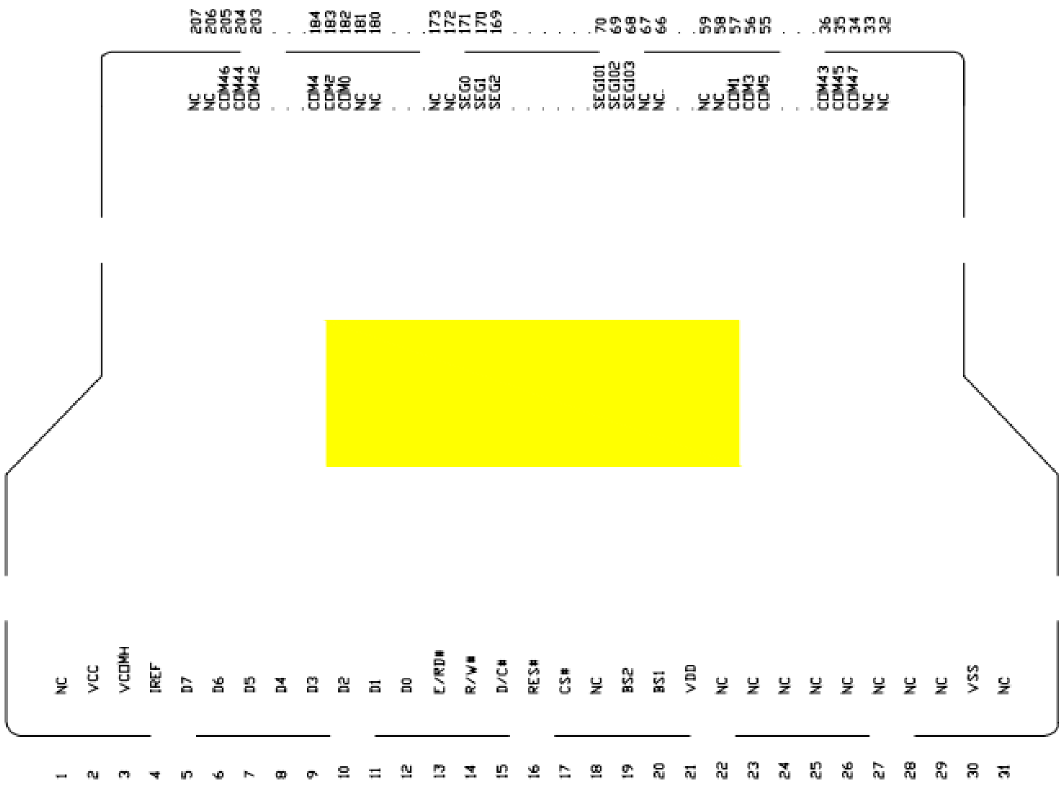


Figure 53 pinout scheme of the SSD1306 controller [23]

I2C interface protocol:

To connect OLED Display with the ESP32 microprocessor we are using the I2C protocol with Arduino, so a brief introduction to his protocol will be needed:

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD, but in our particular case we are only the I2C protocol is going to connect just two things, the ESP32 and the SSD1306 controller [24].

Like UART communication, I2C only uses two wires to transmit pure data and commands between two devices: SDA and SCL, but it also needs two more wires for control:

SA0: slave address bit, to recognise the slave address before the transmission, but as we had said earlier we are connection just two things, so we won´t need this wire. This one is needed when we have a bunch of slaves or masters.

RST or RES#: is used for the initialization of device before start transmitting its need to restart de device and his parameters.

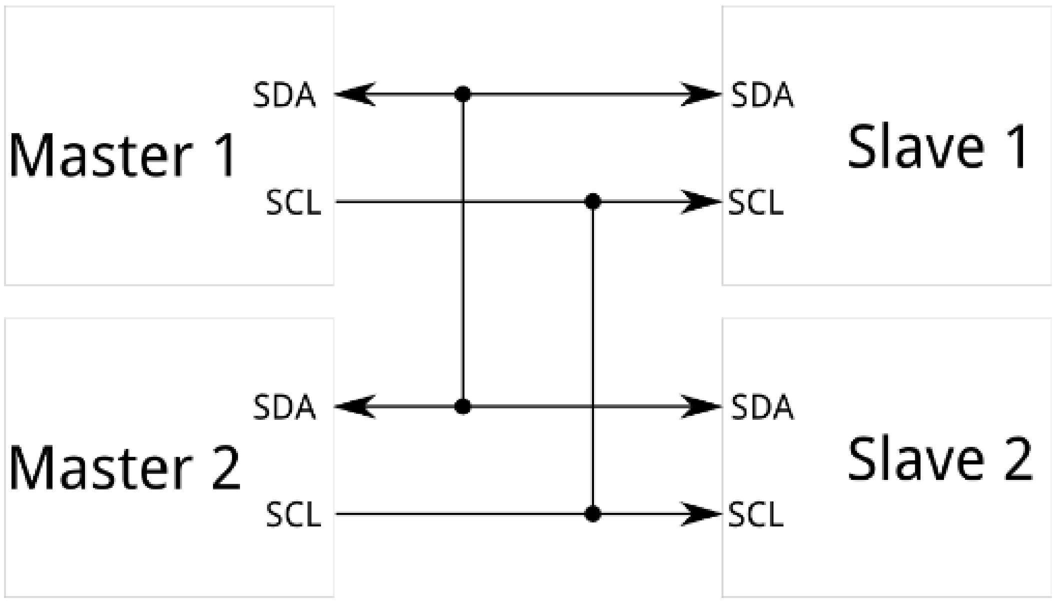


Figure 54 Master-Slave communication with I2C interface basic diagram [24]

SDA (Serial Data) – The I2C bus data signal, communication channel between master and slave, not just data, also ACK is sent through that.

In this bus we have two wires tied together: the SDAin and SDAout. Through SDAin pin the data its transmitted, so as to it it must be always connected, besides the SDAout its just for the ACK so it may be disconnected, losing the ACK message and letting our communication as unidirectional.

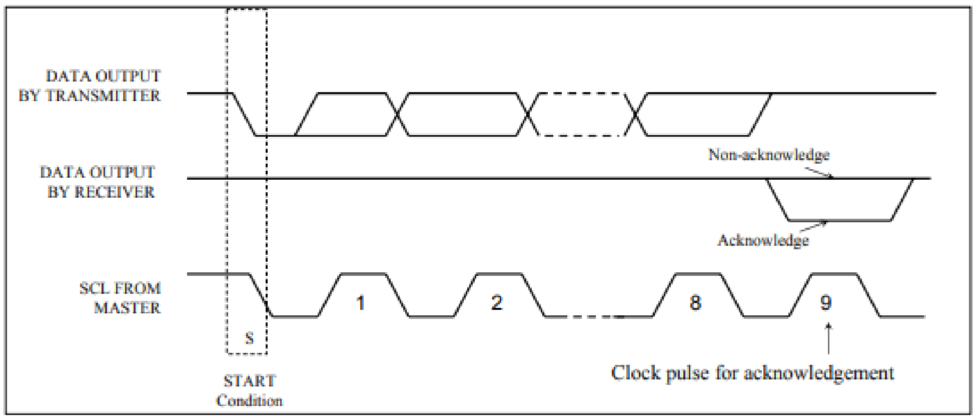


Figure 55 ACK message example [23]

SCL (Serial Clock) – The line that carries the clock signal. Each transmission of data bit takes place during a single clock period of SCL.

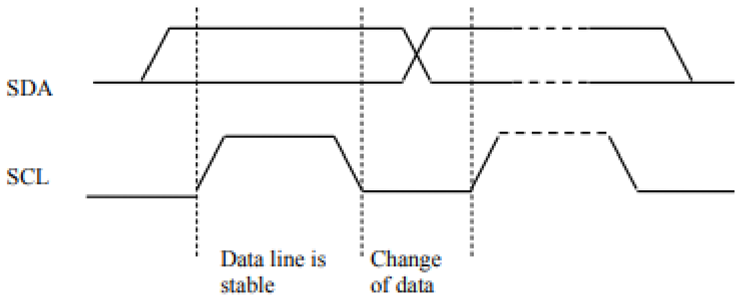


Figure 56 example of the SCL functioning [23]

Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line).

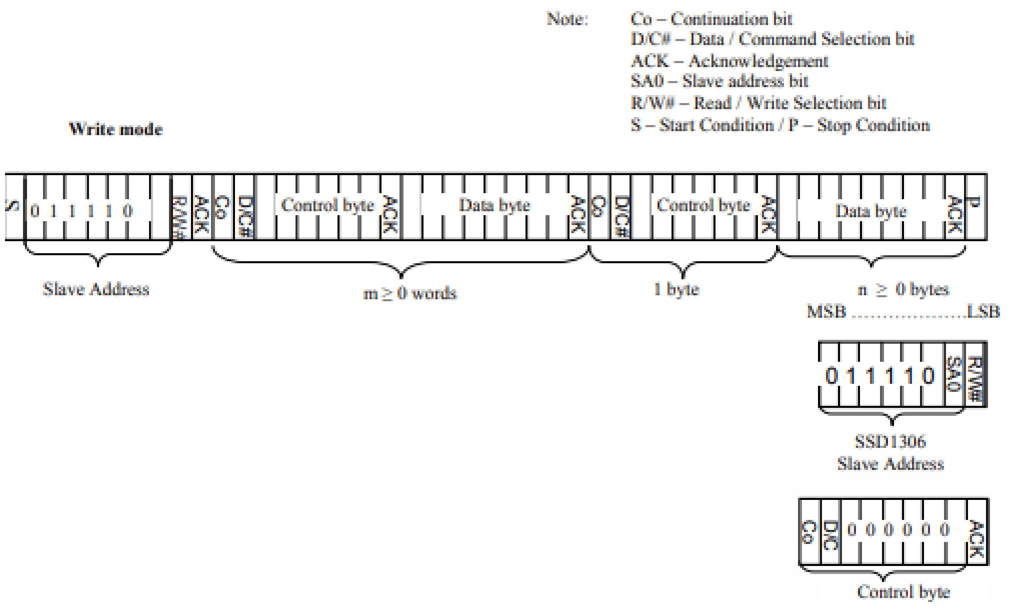


Figure 57 I2C bus data format in the write mode [23]

As we are going to use a I2C protocol to communicate the OLED Display and the ESP32 microprocessor, the pin mapping will be according to the OLED_LoRa_sender.ino code (first board test), (<u>Remember we are using V1</u>):

Definition name in arduino	Pin mapping	Type of the pin in TTGO Lora Module
OLED_SCL	Pin 15	GPIO 5
OLED_SDA	Pin 4	GPIO 19
OLED_RST	Pin 16	GPIO 16

Table 6 pin connection of SSD1306 with ESP32

3.5.3. Programming

The first thing we need to start writing code to control the SSD1306 is to include his own library:

```
#include "SSD1306.h"
```

This is one of the internal sub-libraries of the OLED library; this library includes a lot of different functions to draw circles, lines, squares, to write words and so on.

To start working with this lib, first we have to initialize our OLED display for the connection with esp32. This is possible with:

```
SSD1306Wire (uint8_t _address, uint8_t _sda, uint8_t _scl);
```

This is how the function is initialized with his parameters in the library.

To call and use it in our script:

```
Address = 0x3c

OLED_SDA = 4

OLED_SCL = 15

SSD1306  display (0x3c, 4, 15);
```

After that we can start writing our code and using the different display functions, that we can find in the annexes. It´s also important to say that this is just one of the possible libraries that we may use for the SSD1306 controller.

3.6. The schematic diagram design (including internal connections of presented TTGO Lora module).

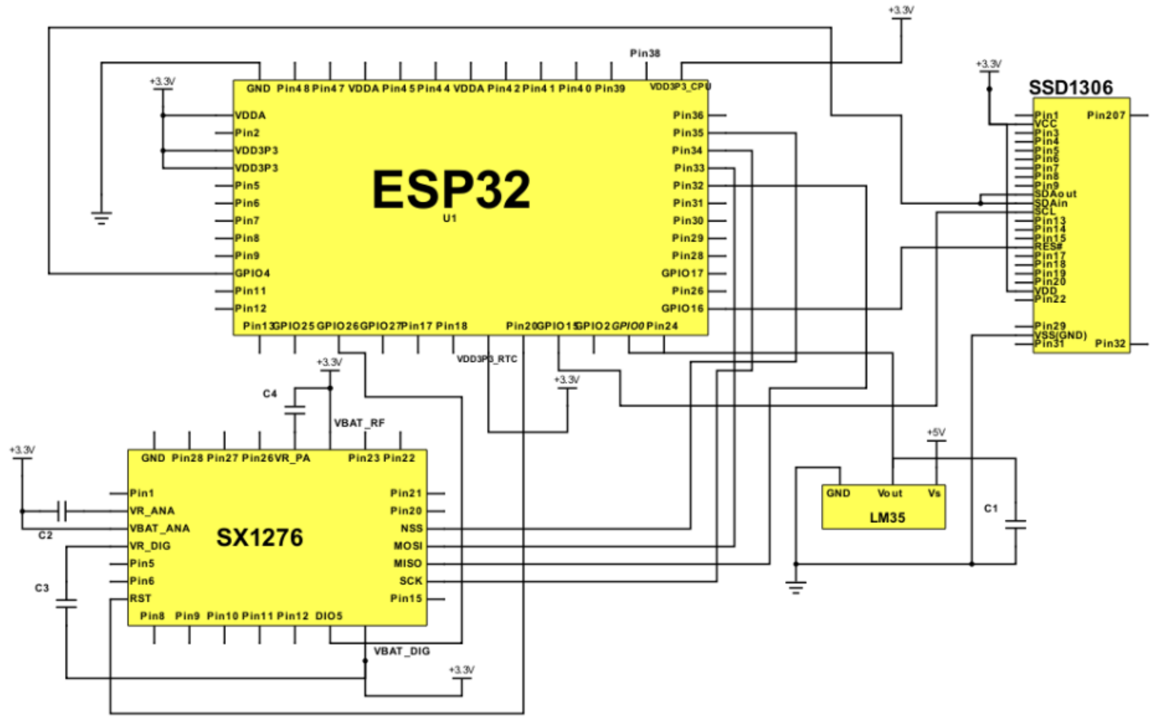


Figure 58 Schematic of TTGO LoRa module

3.7. The data reading and transmission flowchart design.

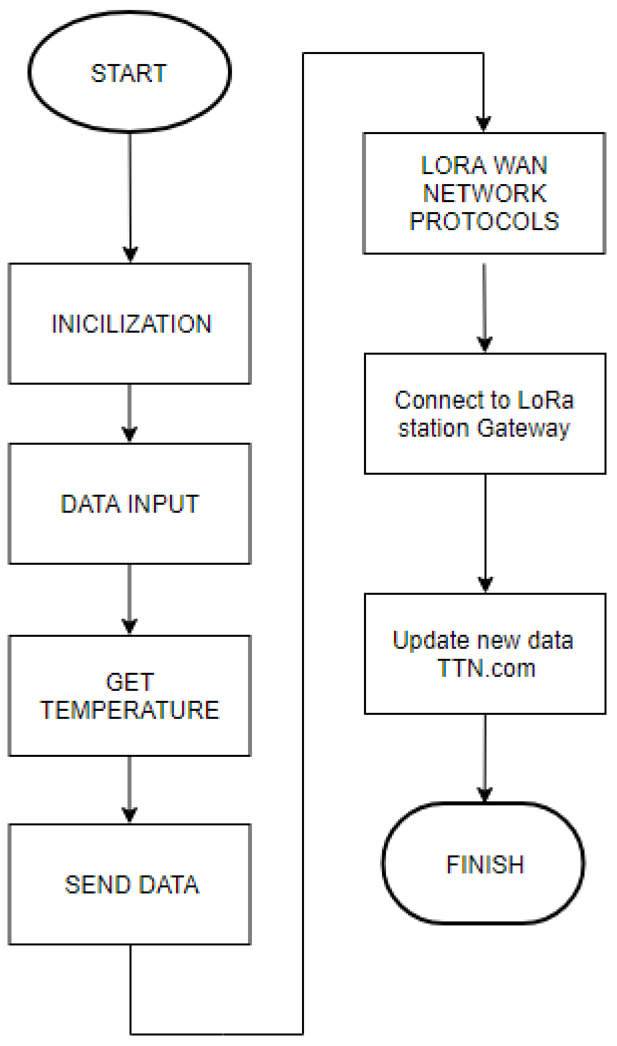


Figure 59 Global flowchart for the LoRa WAN project

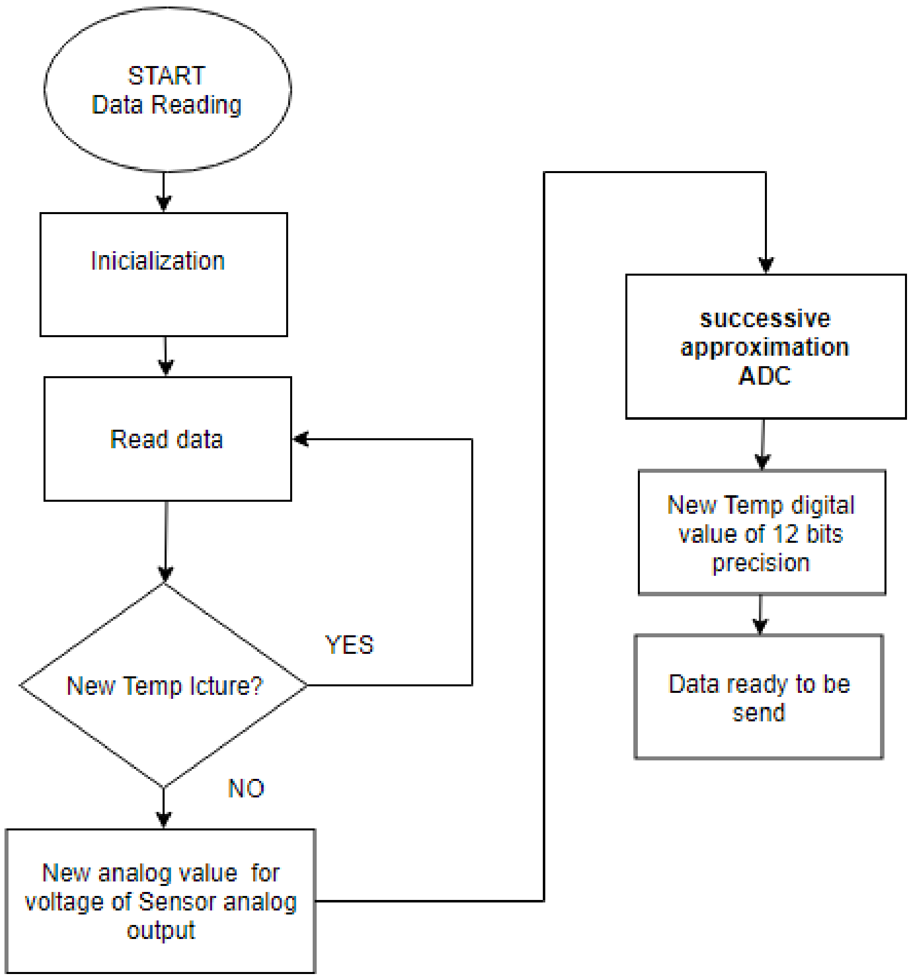

START
Data Reading
Inicialization
Read data
New Temp Icture?
YES
NO
New analog value for voltage of Sensor analog output
successive approximation ADC
New Temp digital value of 12 bits precision
Data ready to be send

Figure 60 Data Reading flowchart for the LoRa WAN project

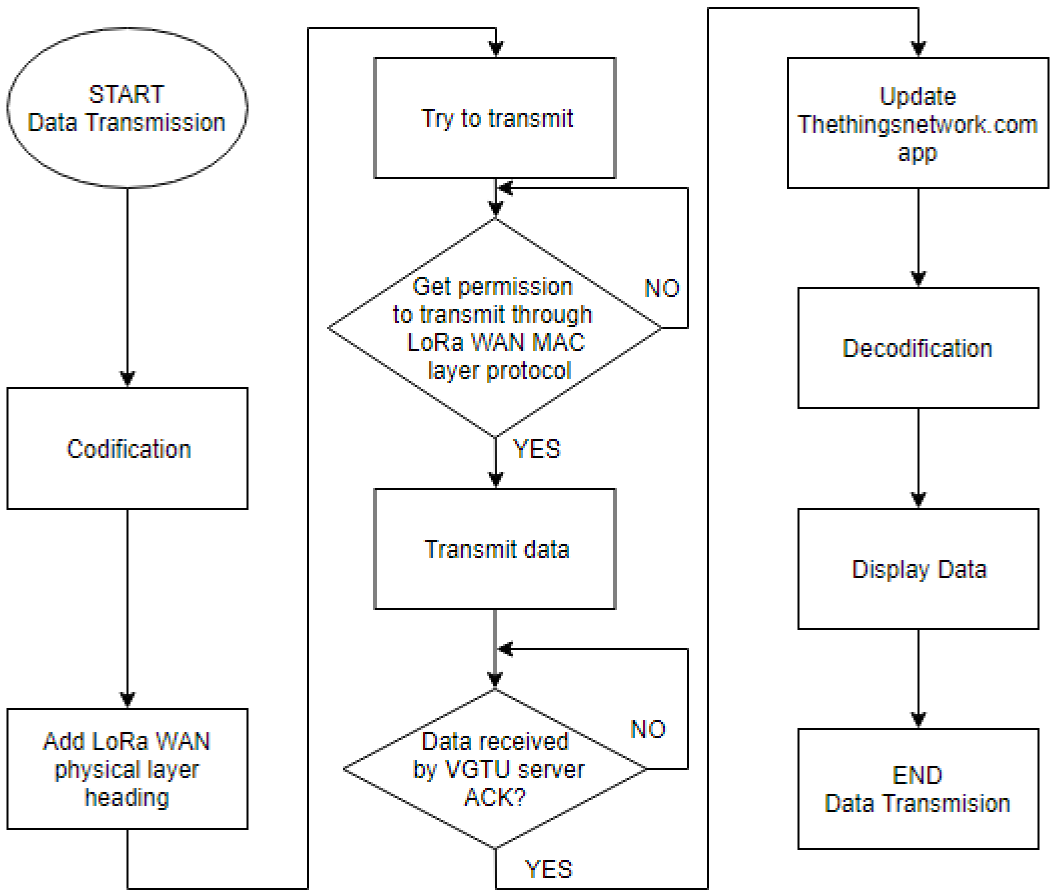

START
Data Transmission
Codification
Add LoRa WAN physical layer heading
Try to transmit
Get permission to transmit through LoRa WAN MAC layer protocol
NO
YES
Transmit data
Data received by VGTU server ACK?
NO
YES
Update Thethingsnetwork.com app
Decodification
Display Data
END
Data Transmision

Figure 61 Data Transmission Flowchart of the LoRa WAN project

3.8. The program design using Arduino shield with libraries.

3.8.1. OLED_LoRa_sender.ino

Libraries and definitions:

1-SPI.h:

It's an Arduino proper library, that includes all the commands to perform a proper SPI devices transmission.

Function examples:

SPI.begin(); Initializes the SPI bus by setting SCK, MOSI, and SS to outputs, pulling SCK and MOSI low, and SS high.

2-LoRa.h:

This Arduino library is design for communications and transmission between two devices using LoRa technology, just sending and receiving unformatted bits between them. Not even the payload format in a LoRa WAN is used here. This library provides us an interface between our MCU and the LoRa transceiver to then transmit our data with its functions LoRa.beginPacket(), LoRa.print(), LoRa.endPacket().

Functions examples:

LoRa.setPins(SS,RST,DI0); It’s easy to realise that this function will set the I2C communication pins between our transceiver and our MCU, and initialize it.

3-Wire.h:

This library allows you to communicate with I2C / TWI devices. In our particular case we will need to manage the transmissions between the SSD1306 controller and the ESP32 MCU. We won´t use wire methods on this script, it will be used by the other library, the one that mange the OLED display, the SSD1306, but this library has been explained in previous points.

```
#include <SPI.h>

#include <LoRa.h>

#include <Wire.h>

#include "SSD1306.h"

#include "images.h"
```

Define all the pin connections that are needed to be made, for the SPI communication between SX1276 and ESP32.

```
#define SCK     5    // GPIO5  -- SX1278's SCK

#define MISO    19   // GPIO19 -- SX1278's MISO

#define MOSI    27   // GPIO27 -- SX1278's MOSI

#define SS      18   // GPIO18 -- SX1278's CS
```

```
#define RST     14   // GPIO14 -- SX1278's RESET

#define DI0     26   // GPIO26 -- SX1278's IRQ(Interrupt Request)

#define BAND    868E6
```

Initializations and declarations:

`unsigned int counter = 0;` A simple counter that will be increasing its value to see how the other lora ttgo module receive it and display it correctly.

V1 modification in the ssd1306 controller pin connections for the I2C communication

```
SSD1306 display(0x3c, 4, 15);

String rssi = "RSSI --";

String packSize = "--";

String packet ;
```

Void_setup():

```
void setup() {

  pinMode(16,OUTPUT);

  pinMode(2,OUTPUT);


  digitalWrite(16, LOW);    // set GPIO16 low to reset OLED

  delay(50);
```

```
    digitalWrite(16, HIGH); // while OLED is running, must set
  GPIO16 in high

    Serial.begin(9600);
    while (!Serial);
    Serial.println();
    Serial.println("LoRa Sender Test");
```

`SPI.begin(SCK,MISO,MOSI,SS);` The function to setup and initialize the SPI communications

`LoRa.setPins(SS,RST,DI0);` The function to setup and initialize the LoRa communications

`if (!LoRa.begin(868E6)) {` To start LoRa technology is so important to put the European frequency band

```
    Serial.println("Starting LoRa failed!");
    while (1);
  }
```

`//LoRa.onReceive(cbk);` these two lines are commented because are for the receiver version

```
  //LoRa.receive();


  Serial.println("init ok");
```

`display.init();` Initialization of the oled display to start running

```
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  logo();
  delay(1500);

}
```

Void loop():

```
void loop() {

  display.clear();

  display.setTextAlignment(TEXT_ALIGN_LEFT);

  display.setFont(ArialMT_Plain_10);



  display.drawString(0, 0, "Sending packet: ");

  display.drawString(90, 0, String(counter));

 display.display();
```

`// send packet`, This four commands form the sending “protocol” , just in a metaphorical way

`LoRa.beginPacket();` This one set the start of the package to send

`LoRa.print("hello ");` This case we are sending a simple hello + counter number

`LoRa.print(counter);`

`LoRa.endPacket();` This one set the start of the package to send

```
  counter++;

  digitalWrite(2, HIGH);    // turn the LED on (HIGH is the voltage
level)

  delay(1000);                          // wait for a second

  digitalWrite(2, LOW);     // turn the LED off by making the voltage
LOW

  delay(1000);a                         // wait for a second

}
```

3.8.2. Ttgo_abp_lorawan.ino

Lmic.h lib explanation:

The LMiC library [25] gives us an event-based programming model where all protocol events are dispatched to the application's onEvent() call back function the one that takes care of the each instant running event. The timings or interrupts are carried by the built-in libraries that handles the run-time environment, taking care pf timer queues and job management.

The code is run in jobs which are executed on the main thread by the main scheduler function os_runloop(). These so-called jobs are coded as regular C functions and will be managed by the run time functions, the ones that control the run-time environment with his initialization, scheduling and execution of the run-time jobs.

For the job management an additional per job control struct osjob_t is required which identifies the job and stores all the information needed to manage and handle it. The jobs should only update state and schedule actions, which will trigger new job or event call-backs.

All an application has to do is to initialize the run-time environment using the os_init() function and to run the job scheduler function os_runloop(), which does not return. In order to launch all the protocol actions and generate events, an initial job needs to be set up. Therefore, a start-up job is scheduled using the os_setCallback() function, in our particular case we are using the os_setTimedCallback() function, which differs with the other in the time when job is going to be prepared to run. The first job prepares a immediately runnable job and the second one scheduled a timed job fixed with the absolute system time.

Hal/hal.h lib explanation:

This library is a hardware abstraction layer that treats and handle low level data and information more related to the hardware drivers and its management.

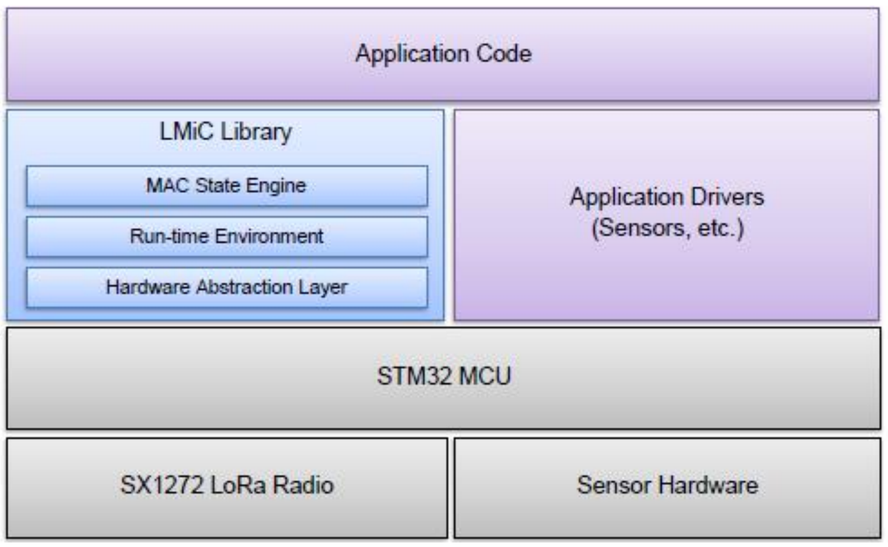


Figure 62 Diagram of the place that Hardware abstraction layer takes up in the LMIC library and that on the LoRa WAN architecture [25]

To understand how this library works we are going to review a couple of examples of the Hal library functions:

`void hal_init ()`: Initialize the hardware abstraction layer and configure the components for: IO, SPI, TIMER, IRQ for other hal functions.

`void hal_failed ()`: Perform a fatal failure situation resetting everything to a safe state position.

`void hal_pin_rxtx (u1_t val)`: Drive the digital output pins RX and TX (0=receive, 1=transmit) getting them prepared to transmit.

U8x8lib.h lib explanation:

This lib it's so similar to the SSD1306 library we use previously on the lora sender and receiver example, we use this now because it's the one that had been tested for the LoRa WAN protocol. It has typical display controllers functions like:

```
u8x8.setCursor(0, 5);
```

The set cursor function just put the location where we are going to start displaying things.

```
u8x8.printf("TIME %lu", os_getTime());
```

The printf function works exactly equal to the c++ function, it simply displays some characters or numbers.

```
u8x8.drawString(0, 7, "EV_SCAN_TIMEOUT");
```

The drawstring function is so similar to printf plus set cursor, because it select the starting position to display and the string to display.

Arduino code:

```
#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>
#include <U8x8lib.h>

#define BUILTIN_LED 25

// the OLED used
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(/* clock=*/ 15, /*
data=*/ 4, /* reset=*/ 16);
```

The OLED controller pin connections for the I2C communication with the ESP32.

```
// LoRaWAN NwkSKey, network session key
```

This is the default Semtech key, which is used by the early prototype TTN network.

```
static const PROGMEM u1_t NWKSKEY[16] = { 0x75, 0x15, 0x56,
0x54, 0x94, 0xD7, 0xCD, 0x35, 0x26, 0x04, 0x61, 0x7A, 0xC9,
0xB5, 0xFC, 0x92 };

// LoRaWAN AppSKey, application session key
```

This is the default Semtech key, which is used by the early prototype TTN network.

```
static const u1_t PROGMEM APPSKEY[16] = { 0xAA, 0x3F, 0xAA,
0xFD, 0xCE, 0x05, 0x77, 0xAA, 0xF0, 0x7C, 0x2A, 0x66, 0xCC,
0x6B, 0x7E, 0xAD };

// LoRaWAN end-device address (DevAddr)
```

`static const u4_t DEVADDR = 0x26011467;` This address has to be changed for every node!

These callbacks are only used in over-the-air activation, so they are left empty here (we cannot leave them out completely unless DISABLE_JOIN is set in config.h, otherwise the linker will complain).

```
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }
```

`static uint8_t mydata[] = "Hello, world!";` This is going to be the data that we are going to send

`static osjob_t sendjob;` the osjob_t is a structure that identifies the job and stores context information, our job will be called sendjob.

Schedule TX every “n” seconds (might become longer due to duty cycle limitations).

```
const unsigned TX_INTERVAL = 30;
```

Pin mapping, to communicate the SX1276 with the ESP32 through the SPI protocol.

```
const lmic_pinmap lmic_pins = {
  .nss = 18, This is the salve selection pin
  .rxtx = LMIC_UNUSED_PIN,
  .rst = 14, This is the reset pin, RST.
  .dio = {26, 33, 32}, This are going to be the MISO, MOSI and
IRQ pins.
};
```

void onEvent (ev_t ev):

The implementation of this call back function may react on certain events and trigger new actions based on the event and the LMIC state, it works like kind of conventional state-machine. Typically, an implementation processes the events it is interested in and schedules further protocol actions using the LMIC API. The following events will be reported:

```
void onEvent (ev_t ev) {
    Serial.print(os_getTime());
    u8x8.setCursor(0, 5);
    u8x8.printf("TIME %lu", os_getTime());
    Serial.print(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            Serial.println(F("EV_SCAN_TIMEOUT"));
            u8x8.drawString(0, 7, "EV_SCAN_TIMEOUT");
            break;
        case EV_BEACON_FOUND:
            Serial.println(F("EV_BEACON_FOUND"));
            u8x8.drawString(0, 7, "EV_BEACON_FOUND");
            break;
        case EV_BEACON_MISSED:
            Serial.println(F("EV_BEACON_MISSED"));
            u8x8.drawString(0, 7, "EV_BEACON_MISSED");
            break;
        case EV_BEACON_TRACKED:
            Serial.println(F("EV_BEACON_TRACKED"));
            u8x8.drawString(0, 7, "EV_BEACON_TRACKED");
            break;
        case EV_JOINING:
            Serial.println(F("EV_JOINING"));
            u8x8.drawString(0, 7, "EV_JOINING");
            break;
        case EV_JOINED:
            Serial.println(F("EV_JOINED"));
            u8x8.drawString(0, 7, "EV_JOINED ");
```

Disable link check validation (automatically enabled during join, but not supported by TTN at this time).

`LMIC_setLinkCheckMode(0);` Enable/disable link check validation. Link check mode is enabled by default and is used to periodically verify network connectivity. Must be called only if a session is established. In this case is disabled.

```
                break;
            case EV_RFU1:
                Serial.println(F("EV_RFU1"));
                u8x8.drawString(0, 7, "EV_RFUI");
                break;
            case EV_JOIN_FAILED:
                Serial.println(F("EV_JOIN_FAILED"));
                u8x8.drawString(0, 7, "EV_JOIN_FAILED");
                break;
            case EV_REJOIN_FAILED:
                Serial.println(F("EV_REJOIN_FAILED"));
                u8x8.drawString(0, 7, "EV_REJOIN_FAILED");
                break;
            case EV_TXCOMPLETE:
```

The data prepared via LMIC_setTxData2() has been sent, and eventually downstream data has been received in return. If confirmation was requested, the acknowledgement has been received.

```
                Serial.println(F("EV_TXCOMPLETE (includes waiting
for RX windows)"));
                u8x8.drawString(0, 7, "EV_TXCOMPLETE");
                digitalWrite(BUILTIN_LED, LOW);
                if (LMIC.txrxFlags & TXRX_ACK)
```

If we have flags and confirmed UP frame was acked

```
                  Serial.println(F("Received ack"));
                  u8x8.drawString(0, 7, "Received ACK");
                if (LMIC.dataLen) {
                  Serial.println(F("Received "));
                  u8x8.drawString(0, 6, "RX ");
                  Serial.println(LMIC.dataLen);
```

LMIC.dataLen is one of the flag binary comands nad part of the LIMIC structure, // 0 no data or zero length data, >0-byte count of data

```
                  u8x8.setCursor(4, 6);
                  u8x8.printf("%i bytes", LMIC.dataLen);
                  Serial.println(F(" bytes of payload"));
```

```
                    u8x8.setCursor(0, 7);
                    u8x8.printf("RSSI %d SNR %.1d", LMIC.rssi,
LMIC.snr);
```

Received Signal Strength Indicator, the RSSI intensity of the signal and the SNR signal-to-noise ratio the relation between the signal and the noise.

```
                }
```

Schedule next transmission

```
                os_setTimedCallback(&sendjob,
os_getTime()+sec2osticks(TX_INTERVAL), do_send);
                break;
            case EV_LOST_TSYNC:
                Serial.println(F("EV_LOST_TSYNC"));
                u8x8.drawString(0, 7, "EV_LOST_TSYNC");
                break;
            case EV_RESET:
                Serial.println(F("EV_RESET"));
                u8x8.drawString(0, 7, "EV_RESET");
                break;
            case EV_RXCOMPLETE:
                // data received in ping slot
                Serial.println(F("EV_RXCOMPLETE"));
                u8x8.drawString(0, 7, "EV_RXCOMPLETE");
                break;
            case EV_LINK_DEAD:
                Serial.println(F("EV_LINK_DEAD"));
                u8x8.drawString(0, 7, "EV_LINK_DEAD");
                break;
            case EV_LINK_ALIVE:
                Serial.println(F("EV_LINK_ALIVE"));
                u8x8.drawString(0, 7, "EV_LINK_ALIVE");
                break;
             default:
                Serial.println(F("Unknown event"));
                u8x8.setCursor(0, 7);
                u8x8.printf("UNKNOWN EVENT %d", ev);
                break;
        }
    }
```

Do_send()function:

The main function into it is the LMIC_setTXData2(), this function will be the main factor to send our package.

```
    void do_send(osjob_t* j){
        // Check if there is not a current TX/RX job running
        if (LMIC.opmode & OP_TXRXPEND) {
            Serial.println(F("OP_TXRXPEND, not sending"));
            u8x8.drawString(0, 7, "OP_TXRXPEND, not sent");
        } else {
            // Prepare upstream data transmission at the next
    possible time.
```

The pure “sending” or transmitting LMIC_setTxData2 (u1_t port, xref2u1_t data, u1_t dlen, u1_t confirmed)

We are using port 1, sending data will be mydata (helloworld string), the length of mydata -1 bit, and no confirmation from receiver (the meaning of the last number wich can take 0 or 1 value).

```
            LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);
            Serial.println(F("Packet queued"));
            u8x8.drawString(0, 7, "PACKET QUEUED");
        }
        // Next TX is scheduled after TX_COMPLETE event.
    }
```

Void setup():

```
    void setup() {
        Serial.begin(115200);
        Serial.println(F("Starting"));
```

```
u8x8.begin();
u8x8.setFont(u8x8_font_chroma48medium8_r);
u8x8.drawString(0, 1, "LoRaWAN LMiC TTN Node...");

SPI.begin(5, 19, 27);

// LMIC init
```

`os_init();` Initialize the run-time environment

Reset the MAC state. Session and pending data transfers will be discarded.

`LMIC_reset();` A must use function to start a loraWAN network transmission, it’s a part of the LMIC initialization needed

Set static session parameters. Instead of dynamically establishing a session by joining the network, precomputed session parameters are being provided.

```
#ifdef PROGMEM
```

On AVR, these values are stored in flash and only copied to RAM once. Copy them to a temporary buffer here, LMIC_setSession will copy them into a buffer of its own again.

```
uint8_t appskey[sizeof(APPSKEY)];
uint8_t nwkskey[sizeof(NWKSKEY)];
memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
#else
```

If not running an AVR with PROGMEM, just use the arrays directly

```
LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
#endif

#if defined(CFG_eu868)
```

Set up the channels used by the Things Network, which corresponds to the defaults of most gateways. Without this, only three base channels from the LoRaWAN specification are used, which certainly works, so it is good for debugging, but can overload those frequencies, so be sure to configure the full your network here (unless your network autoconfigures them). Setting up channels should happen after LMIC_setSession, as that configures the minimal channel set.

```
        // NA-US channels 0-71 are configured automatically
    //        LMIC_setupChannel(0,  868100000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
    //        LMIC_setupChannel(1,  868300000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7B), BAND_CENTI);      // g-band
    //        LMIC_setupChannel(2,  868500000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
    //        LMIC_setupChannel(3,  867100000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
    //        LMIC_setupChannel(4,  867300000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
    //        LMIC_setupChannel(5,  867500000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
    //        LMIC_setupChannel(6,  867700000,  DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
        LMIC_setupChannel(7,     867900000,     DR_RANGE_MAP(DR_SF12,
    DR_SF7),  BAND_CENTI);      // g-band
    //         LMIC_setupChannel(8,  868800000,  DR_RANGE_MAP(DR_FSK,
    DR_FSK),  BAND_MILLI);      // g2-band
    //
```

TTN defines an additional channel at 869.525Mhz using SF9 for class B devices' ping slots. LMIC does not have an easy way to define set this frequency and support for class B is spotty and untested, so this frequency is not configured here.

```
        #elif defined(CFG_us915)
```

NA-US channels 0-71 are configured automatically but only one group of 8 should (a subband) should be active TTN recommends the second sub band, 1 in a zero-based count.

```
        LMIC_selectSubBand(1);
        #endif

        // Disable link check validation
        LMIC_setLinkCheckMode(0);
        // TTN uses SF9 for its RX2 window.
        LMIC.dn2Dr = DR_SF9;
```

Set data rate and transmit power for uplink (note: txpow seems to be ignored by the library and to not have a great influence at the hardware level)

```
        LMIC_setDrTxpow(DR_SF7,14);

        // Start job
        do_send(&sendjob);
    }
```

Void loop():

Like we have said earlier one of the most important things to do in a LMIC based program is run the job scheduler function os_runloop(), which Execute run-time jobs from the timer and from the run queues. This function is the main action dispatcher. It does not return and must be run on the main thread, in other words, it will be the “brain” of this event based-program.

In our case we are using the is_runloop_once(); as its name said it happens once.

```
    void loop() {
        os_runloop_once();
    }
```

3.8.3. Ttgo_abp_lorawan_LM35sensor2.ino

This code is going to be one to perform our main task, the temperature data transmission to our TTN app through LoRa WAN. This code only defers with the last in a few things: To send the digital values provided by the ESP32 ADC, we must improve the code in order to send more than a “Hello World”. So first we create a new data buffer where the information will be stored before transmission, this buffer is going to be a char of 8 lots:

```
    static uint8_t mydata[] = { 0,0,0,0,0,0,0,0};
```

And we are going to need to use a function to convert the number read into a char, using the dtostrf, similar to well know functions as sprint but more useful for this case:

```
dtostrf(temperature,5,2,(char*)mydata);
```

In addition, in the do_send function we need to change the LMIC_setData2 function parameters for the new buffer structure.

```
LMIC_setTxData2(1, mydata, strlen((char*) mydata), 0);
```

Of course before all this technical modifications we have to introduce the analogRead() with its pin, and the algorithm to transform the raw data into temperature values.

```
float RawValue = analogRead(34);
float temperature;
float Voltage;

Voltage = (RawValue / 4095.0) * 3300; // 5000 to get millivots.
    temperature = (Voltage * 0.1);
```

This new code features are based on: [26]

4.Obtained results and conclusions.

4.1. Sensor and Data Reading and transformation to digital (ADC)

Analog to Digital Converter of the ESP32:

ESP32 integrates two 12-bit SAR ("Successive Approximation Register"): A successive approximation ADC is a type of analog-to-digital converter that converts a continuous analog waveform into a discrete digital representation via a binary search through all possible quantization levels before finally converging upon a digital output for each conversion) ADCs (Analog to Digital Converters) and supports measurements on 18 channels (analog enabled pins). Some of these pins can be used to build a programmable gain amplifier which is used for the measurement of small analog signals.

Successive Approximation ADC:

This is the most commonly used ADC when we need medium or high speed of conversion, between microseconds and decimals of microseconds. A comparator is used to tight up successively a range of input voltage. In each consecutive step the converter compares the input voltage with the output voltage of internal digital to analog converter, which may be the mid-way point in the selected voltage range of values. Each step of this process is stored in a successive approximation register (SAR).

For example, we take input voltage value of 6.3 V, and the initial range is between 0 V and 16 V.

In the first step it compares 6.3 V with 8 V (the mid-way point of the 0 to 16 V range of values). The comparator inform that the input value is smaller than 8V, so as to that the SAR updates the range to a new and tighter 0 to 8 V.

The second step will be similar to the first, despite now the input value is compared to 4 V (the mid-way point of the 0 to 8 V range). The comparator inform that the input value is bigger than 4V, so as to that the SAR updates the range to a new and tighter 4 to 8 V.

Third step will narrow now the range of values from 6 to 8 V, and so on until we get the expected resolution [27].

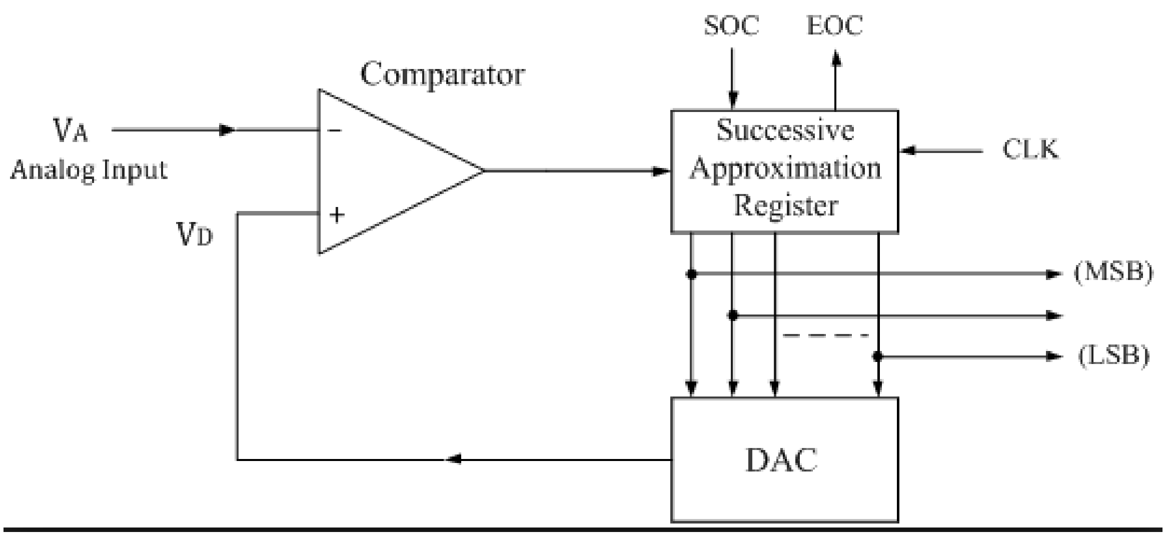


Figure 63 The internal logic circuit of a SAR ADC [27]

The ADC driver API currently only supports ADC1 (9 channels, attached to GPIOs 32-39).

Taking an ADC reading involves configuring the ADC with the desired precision and attenuation settings, and then calling adc1_get_voltage() to read the channel.

It is also possible to read the internal hall effect sensor via ADC1 [28].

4.2. How to use LoRa WAN with TTGO LoRa ESP32

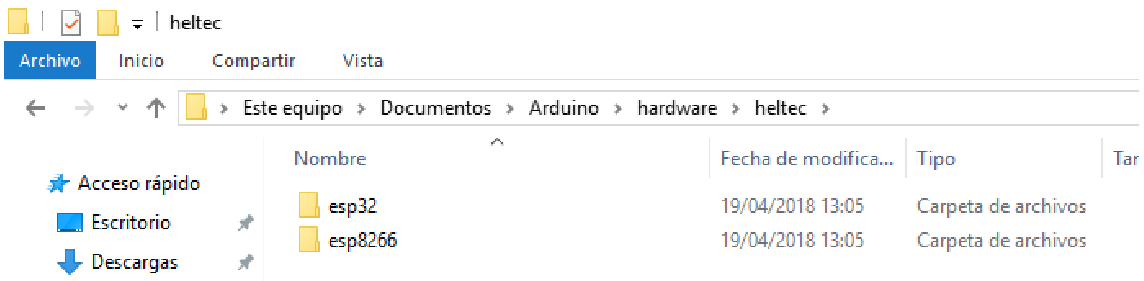
4.2.1. Introduction

In this guide I will explain how I have connected my ttgo lora module v1, to a gateway in our university, working with LoRa WAN, creating an online application for this purpose with thethingsnetwork.org web.

4.2.2. Step by step

Software needs:

Step 1 was making sure that the Arduino IDE becomes capable of compiling code for the board and to upload it to it. To do that you need to go to this repository https://github.com/Heltec-Aaron-Lee/WiFi_Kit_series. Scroll down to get to the English instructions. I didn’t use the option to checkout via git, but simple downloaded it as a Zip file (big green button upper right). In the folder ..\Documents\Arduino\hardware on my laptop I created a folder “heltec” and within that I copied the folders “esp32” and “esp8266” from the ZIP file.



heltec

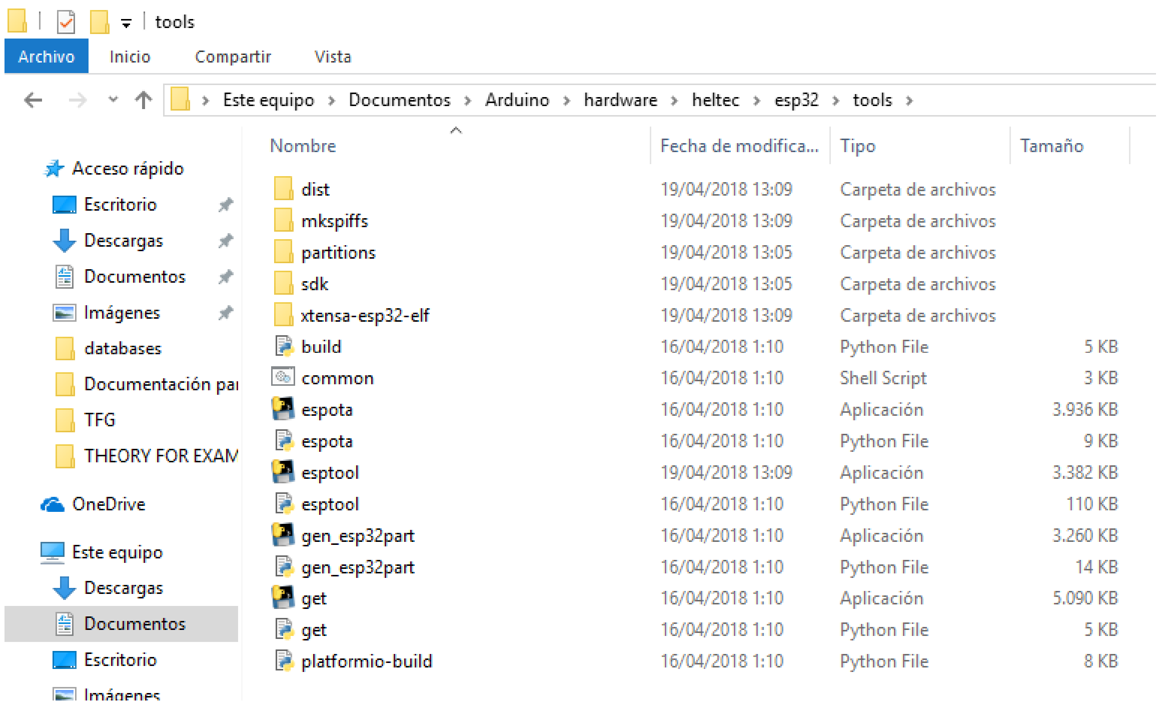
Archivo | Inicio | Compartir | Vista

Este equipo > Documentos > Arduino > hardware > heltec >

Nombre	Fecha de modifica...	Tipo
esp32	19/04/2018 13:05	Carpeta de archivos
esp8266	19/04/2018 13:05	Carpeta de archivos

Figure 64 heltec files

Important: in the folder ..\Documents\Arduino\hardware\heltec\esp32\tools there is a file called get.exe. You need to double click that file so a number of other files can be downloaded.



tools

Archivo | Inicio | Compartir | Vista

Este equipo > Documentos > Arduino > hardware > heltec > esp32 > tools >

Nombre	Fecha de modifica...	Tipo	Tamaño
dist	19/04/2018 13:09	Carpeta de archivos	
mkspiffs	19/04/2018 13:09	Carpeta de archivos	
partitions	19/04/2018 13:05	Carpeta de archivos	
sdk	19/04/2018 13:05	Carpeta de archivos	
xtensa-esp32-elf	19/04/2018 13:09	Carpeta de archivos	
build	16/04/2018 1:10	Python File	5 KB
common	16/04/2018 1:10	Shell Script	3 KB
espota	16/04/2018 1:10	Aplicación	3.936 KB
espota	16/04/2018 1:10	Python File	9 KB
esptool	19/04/2018 13:09	Aplicación	3.382 KB
esptool	16/04/2018 1:10	Python File	110 KB
gen_esp32part	16/04/2018 1:10	Aplicación	3.260 KB
gen_esp32part	16/04/2018 1:10	Python File	14 KB
get	16/04/2018 1:10	Aplicación	5.090 KB
get	16/04/2018 1:10	Python File	5 KB
platformio-build	16/04/2018 1:10	Python File	8 KB

Figure 65 tools file with its content

Step 2 is to download the Arduino-LMIC library that handles the communication with the SX1276 chip on the board. Here again I downloaded the ZIP-file from github. Since this is a library, it needs to be installed from within the Arduino IDE via Sketchs > Include Library > Add .ZIP library.

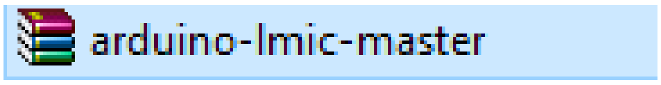
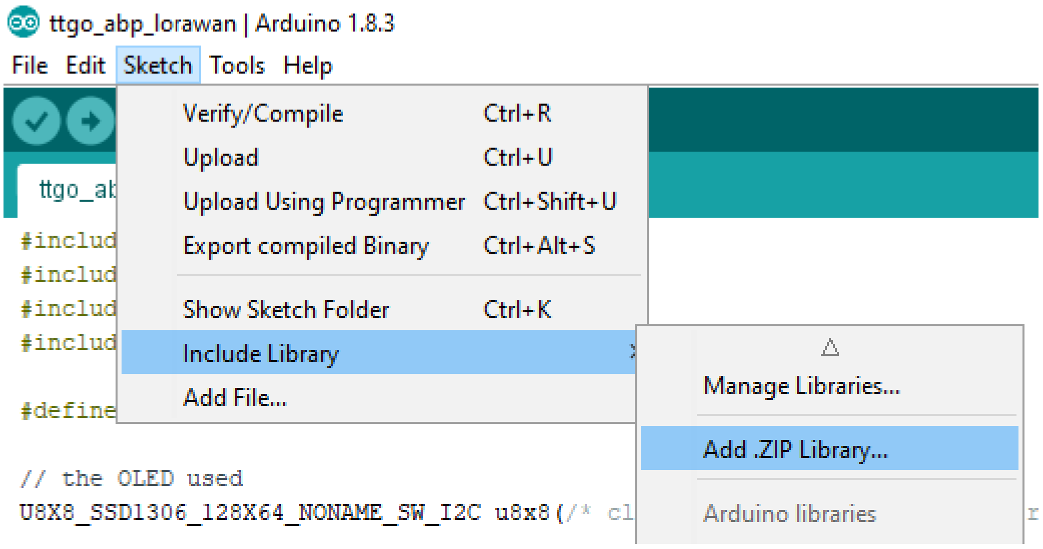


Figure 66 downloaded file with the LMIC library


ttgo_abp_lorawan | Arduino 1.8.3
File Edit Sketch Tools Help
Verify/Compile Ctrl+R
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Export compiled Binary Ctrl+Alt+S
Show Sketch Folder Ctrl+K
Include Library
Add File...
Manage Libraries...
Add .ZIP Library...
Arduino libraries
ttgo_ab

```
#includ
#includ
#includ
#includ

#define

// the OLED used
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(/* cl
```

Figure 67 includeind the LMIC library procedure

Step 3 is making sure we can output stuff to the OLED. For that we need to install the U8x8lib library, (just another different library to control the oled display through the SSD1306 controller, we will use this library because it’s the one that had been checked to work properly with LoRa WAN). I used the build in option to install the library via Sketch > Include Library > Manage Library. Just search for U8x8 and install it.

Now we can start. As board you select “WiFi_LoRa_32”. The LIMC repository on github, provided by Matthijs Kooijman also includes a number of examples for use of the code with The Things Network.

That way you’ll get the needed codes to add in your copy of the code (making sure the data of your device gets delivered at the right location).

¡There is another change we have to make within the provided example code!

You have to change this code snippet:

```
// Pin mapping
const lmic_pinmap lmic_pins = {
.nss = 6,
.rxtx = LMIC_UNUSED_PIN,
.rst = 5,
.dio = {2, 3, 4}, };
```

into:

```
    // Pin mapping
    const lmic_pinmap lmic_pins = {
    .nss = 18,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 14,
    .dio = {26, 33, 32}, };
```

This code we have just downloaded have a little problem, it doesn't use the OLED display, and as we had spent money on it we would like to use it. So, we are going to download another piece of uploaded Arduino code from Github, this time from another direction, it's the same code but with a couple of new lines referred to the OLED display library: https://github.com/PiAir/esp32-sx1276.

In this direction we downloaded the ABP example:

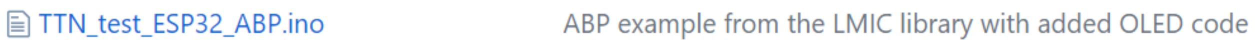

TTN_test_ESP32_ABP.ino   ABP example from the LMIC library with added OLED code

Figure 68 ABP example code that we are going to use

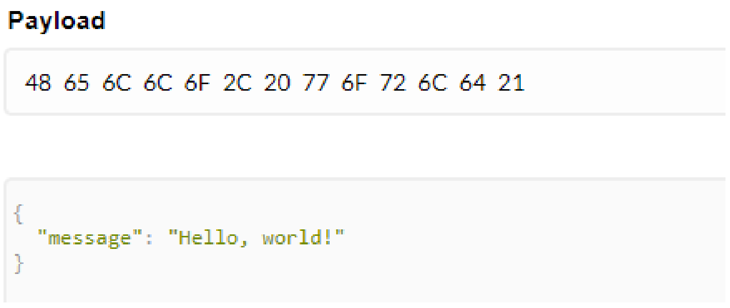
This step by step tutorial is based on: [29].

4.2.3. Decoding of the data transmitted

The example that we are using just keeps sending "Hello, world!" on a loop. If you look for the data received by TTN in the console, you won't see that text there at first. You should see a payload containing "48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21". This has been encoded to its hexadecimal form code, so for example when we see a "48" is the hexadecimal code for "H", the rest of them we can find on the ASCII table. Translating the ASCII values back to a string can be done automatically by TTN if you tell it how to. You can create a payload function for that (you can find it as "Payload Formats" on Application level on our already created application of TTN, we will see how to create it later, not on Device level). If you add this function:

```
function Decoder(bytes, port) {
  var text = String.fromCharCode.apply(null, bytes);
  return {
    message: text
  }
}
```

Then the incoming payload gets translated into a JSON-object containing the string as “message” [30].



Payload

```
48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21
```

```
{
  "message": "Hello, world!"
}
```

Figure 69 Example de decoding.

4.2.4. Attaching the sensor

It’s clear that just sending the same “Hello World” string is not enough, we need now to prepare ourselves to attach a sensor to our TTGO LoRa board and start sending the measured data. To know where to connect that sensor, a schema of the pinouts is needed.

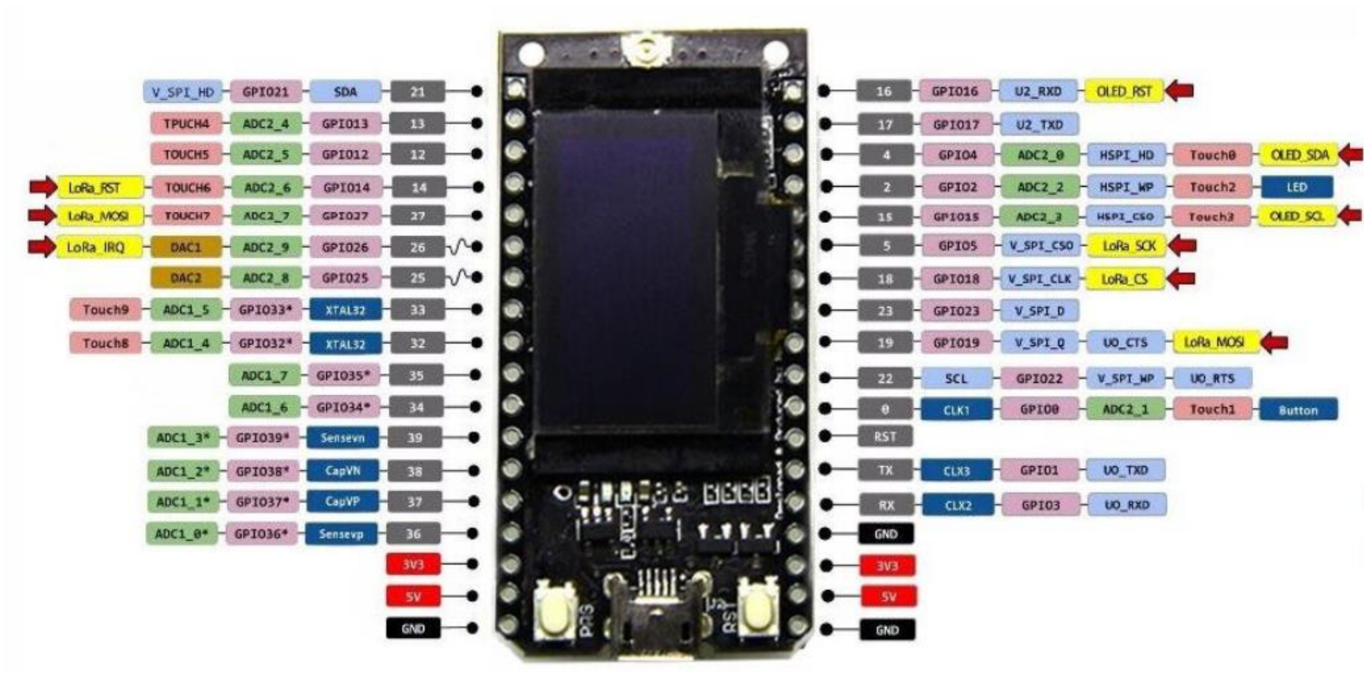

V_SPI_HD GPIO21 SDA 21
TPUCH4 ADC2_4 GPIO13 13
TOUCH5 ADC2_5 GPIO12 12
LoRa_RST TOUCH6 ADC2_6 GPIO14 14
LoRa_MOSI TOUCH7 ADC2_7 GPIO27 27
LoRa_IRQ DAC1 ADC2_9 GPIO26 26
DAC2 ADC2_8 GPIO25 25
Touch9 ADC1_5 GPIO33* XTAL32 33
Touch8 ADC1_4 GPIO32* XTAL32 32
ADC1_7 GPIO35* 35
ADC1_6 GPIO34* 34
ADC1_3* GPIO39* Sensevn 39
ADC1_2* GPIO38* CapVN 38
ADC1_1* GPIO37* CapVP 37
ADC1_0* GPIO36* Sensevp 36
3V3
5V
GND
16 GPIO16 U2_RXD OLED_RST
17 GPIO17 U2_TXD
4 GPIO4 ADC2_0 HSPI_HD Touch0 OLED_SDA
2 GPIO2 ADC2_2 HSPI_WP Touch2 LED
15 GPIO15 ADC2_3 HSPI_CS0 Touch3 OLED_SCL
5 GPIO5 V_SPI_CS0 LoRa_SCK
18 GPIO18 V_SPI_CLK LoRa_CS
23 GPIO23 V_SPI_D
19 GPIO19 V_SPI_Q U0_CTS LoRa_MOSI
22 SCL GPIO22 V_SPI_WP U0_RTS
0 CLK1 GPIO0 ADC2_1 Touch1 Button
RST
TX CLK3 GPIO1 U0_TXD
RX CLK2 GPIO3 U0_RXD
GND
3V3
5V
GND
PRG
RST

Figure 70 TTGO LoRa V1 pinout [19]

Besides needing to have the correct schema, some of the pins are already in use by the existing script. A quick look at the code lists GPIO-pins 15, 4, 16, 18, 14, 26, 33, 32 as already in use. And of course, it wouldn´t be a good idea to use the pins of “LoRa_” and “LED” and “OLED” in the pinout schema, because it makes that I expect more ports to be in use. So that gives me a bunch of free pins to use.

Its important to mark that we are going to measure analog data, so we should use a pin with analog to digital converter, so one of the free pins with the “ADC_” words in it. I will choose the pin 35 as it´s free and has the AD converter ADC1_7 (GPIO35*).

The next step is connecting the analog output of our LM35 to the pin 34, to transform that analog values to digital so as to make it manipulable for the MCU. And finally connect the Vp to the 5V and the GND of our sensor to GND [31].

Programming code for sensing:

At first we can use a typical Arduino function that is also valid for ESP32 ADC, the analogRead(), the argument of this function is the pin where we have attached our sensor; so in our case, it will be analogRead(35), this would give as the value ready to work with it.

Before this we should configure the pin to work as ADC using adcAttachPin(34), on the void setup() [32] .

It’s also important to use the Serial communication, first we configure it in the void setup() electing the baudrate of communication, in our case Serial.begin(115200); after that we should print our info to see if everything its working good with the Serial.println() function. This is just a first approach to use sensors with the ESP32, of course after this first test we should integrate that on our major lora WAN code to send transmit this data instead of the hello world string.

-Potentiometer test:

We are using a B10K potentiometer, with linear characteristic, resistance of 10K, and an angle of rotation of 300º $\pm$ 10º.

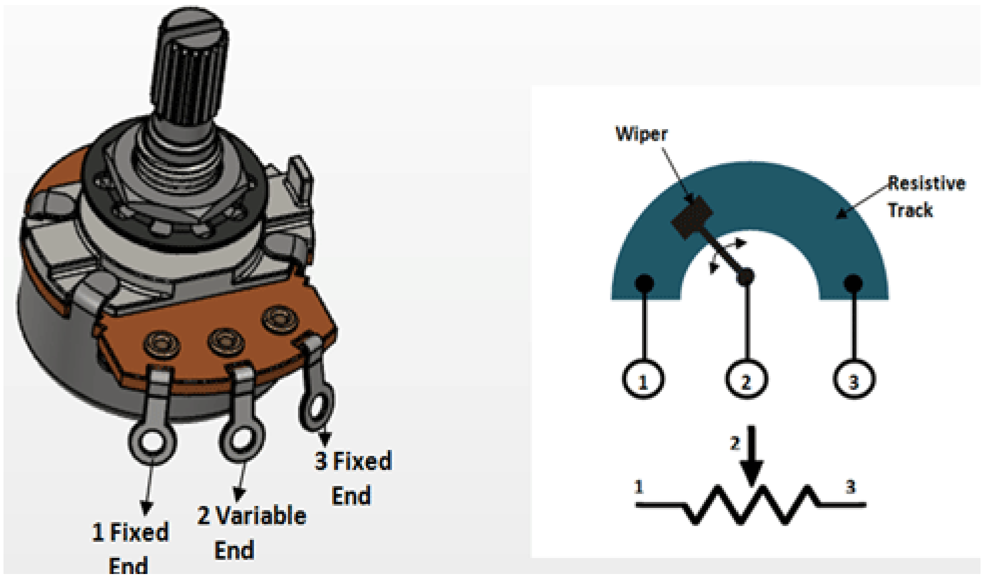
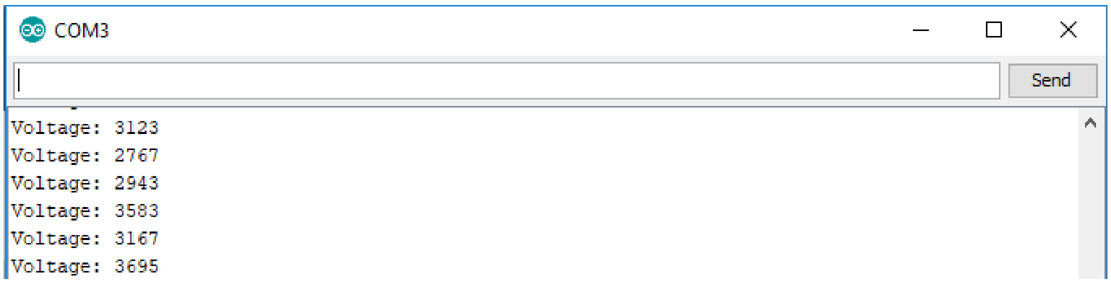


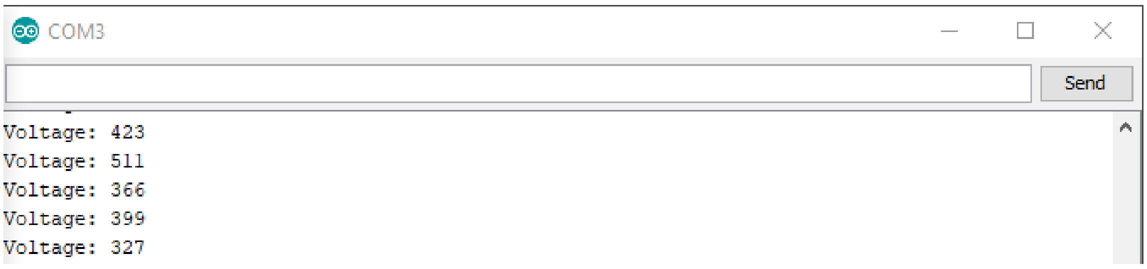
Figure 71 Pinout of the B10K

Figure 72 potentiometer B10K



```
Voltage: 3123
Voltage: 2767
Voltage: 2943
Voltage: 3583
Voltage: 3167
Voltage: 3695
```

Figure 73 Potentiometer analogRead() with 12 bits of resolution, with 5 V



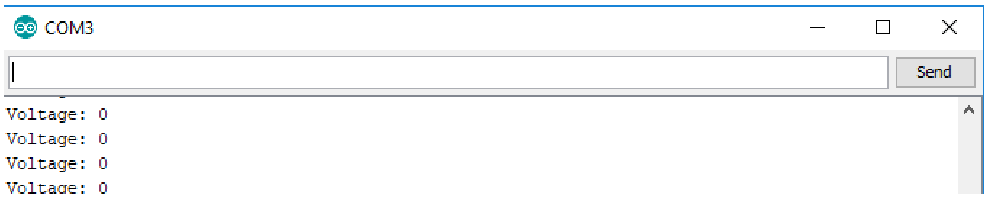
```
Voltage: 423
Voltage: 511
Voltage: 366
Voltage: 399
Voltage: 327
```

Figure 74 Potentiometer analogRead() with 6 bits of resolution with 5 V

We can easily notice that something it´s not working properly because the value of the analogRead() isn´t proportional to the angle of potentiometer´s rotation in all the degrees positions, for example when we are at the 0º the measurement is 0, but if we keep moving until a certain value of rotation the measurement keep on 0º; after that in a short range of degrees (almost 30º) all the range of values appears, from 0 to 4095 (in the 12 bit resolution case), and after that, the 4095 value saturate again, and it´s been held to end of the rotation.

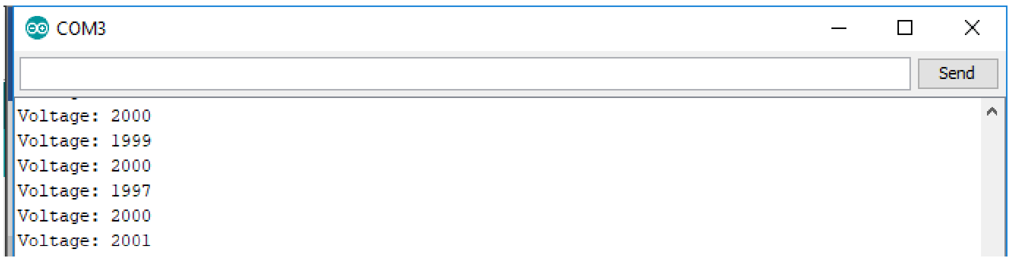
Now I change the V+ value of the potentiometer from 5 V to 3 V. And see what changes:

Finally, we have get the full range of valid values from 0º and 0 V from 5 V values



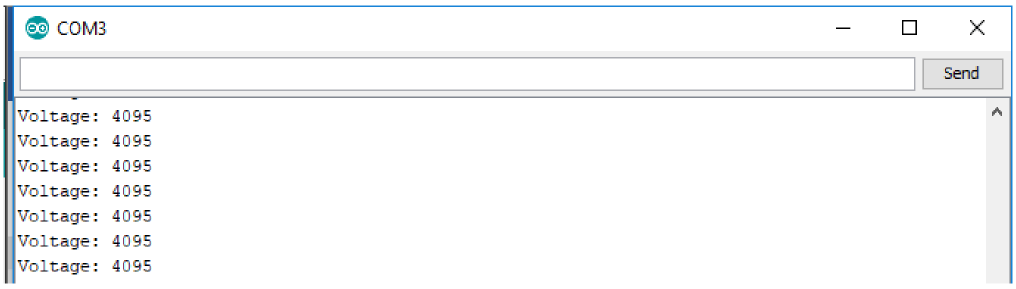
```
Voltage: 0
Voltage: 0
Voltage: 0
Voltage: 0
```

Figure 75 cursor at the start 3.3 V



```
Voltage: 2000
Voltage: 1999
Voltage: 2000
Voltage: 1997
Voltage: 2000
Voltage: 2001
```

Figure 76 cursor at the half way 3.3 V



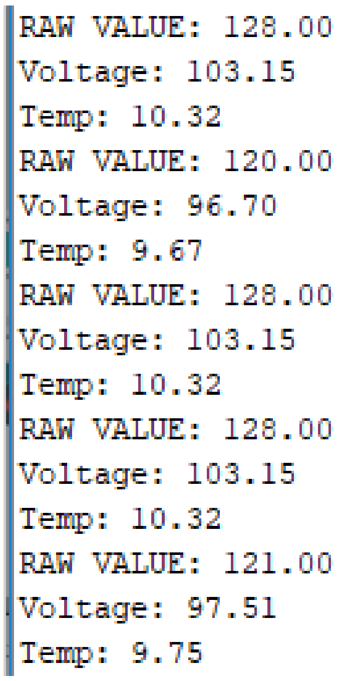
```
Voltage: 4095
Voltage: 4095
Voltage: 4095
Voltage: 4095
Voltage: 4095
Voltage: 4095
Voltage: 4095
```

Figure 77 cursor at the end 3.3 V

Testing the LM35 sensor:

```
Voltage = (RawValue / 4095.0) * 3300; // 3300 to get millivolts.
temperature = (Voltage * 0.1);
```

We use this conversion algorithm because we have a 12-bit resolution, 3300 mV as maximum ADC output value and we have multiplied it by 0.1 (divided it by 10mV), because 10 mV per degree.



```
RAW VALUE: 128.00
Voltage: 103.15
Temp: 10.32
RAW VALUE: 120.00
Voltage: 96.70
Temp: 9.67
RAW VALUE: 128.00
Voltage: 103.15
Temp: 10.32
RAW VALUE: 128.00
Voltage: 103.15
Temp: 10.32
RAW VALUE: 121.00
Voltage: 97.51
Temp: 9.75
```

Figure 78 testing the LM35 at 5V and the 12 bits of resolution

After seeing the results it´s clear that the results of the LM35 sensing test are not what we espected, the temperature it´s not even close to the real one, it´s 26ºC and the sensor gives us around 10ºC.

So now we are going to need to check what is not working. In a quick look we can see that the voltage digital signal provided by the ADC is around 95 mV, and it should be 260 mV aproximately (for the 26ºC); so as to it we checked the voltages with a voltmeter:

- 5 V board power source, its okey its giving 5.14 V
- Analog output of the LM35, its okey is giving us around 267 mV

So now we now that the problem is not due to the power source or the LM35, it is on the ADC of the ESP32. To check the performing accuracy of the ADC we have make a table of measurements, this measurements will be done with the 10K potentiometer to check all posible values, the problems could be on the low voltage levels so we decided to make a measurement each 10 mV in the 0 V to 0.5 V range, then a measurement each 20 mV in the 0.5 V to 1 V, and to finish a measurement each 100 mV (the maximum value its 3.3 V but it has not been necessary to reach it to solve the problem).

The table of measurements is available on the anexes, but for the moment we will check the chart where the VADC values are on the y axes and the Vexp (volts measured with voltmeter) values are on the x axes.

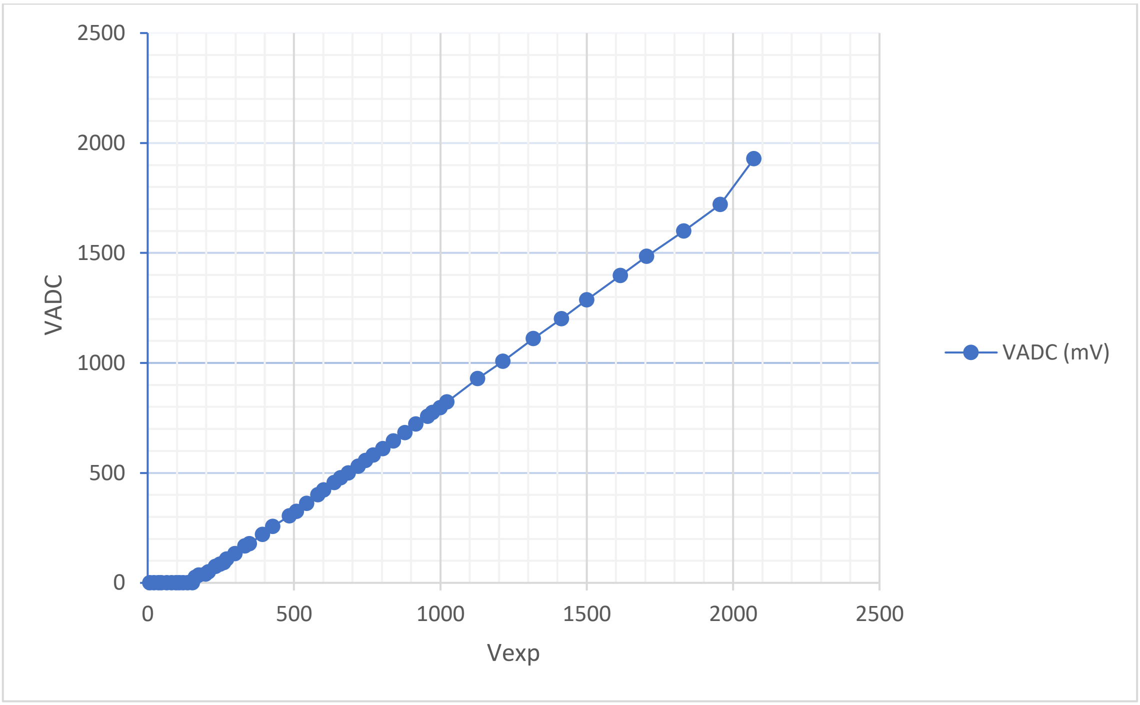


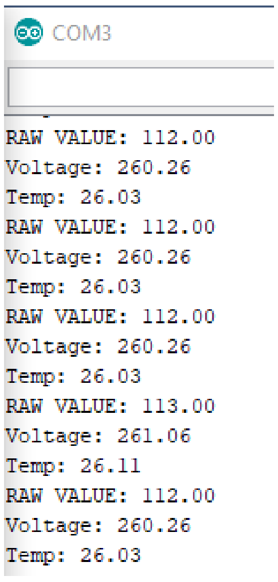
Figure 79 Volts measured versus Volts provided by the ADC

The charts has good linearity response, but now we have found the problem. At low voltages, below 150 mV or 160 mV in particular, the volts of the ADC are always 0 mV. This gives us the key of the problem, we have an offset error. To correct this error, we look at what it may be a common temperature value, 25ºC, so we have a difference of 170 mV between the value measured and the Vadc. If we want to correct it, we have to add the offset value to our algorithm.

Error offset = 170 mV:

```
float voltag = ((temp1*3300)/4095)+170;
```

Now we attach the LM35 back with the board and test it again:


COM3

```
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
RAW VALUE: 113.00
Voltage: 261.06
Temp: 26.11
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
```

Figure 80 Example of values provided by the sensor with the offset error added

The results are the expected, even though we know that this is not a great accuracy sensor. Temperatures of 26ºC are real an acceptable value.

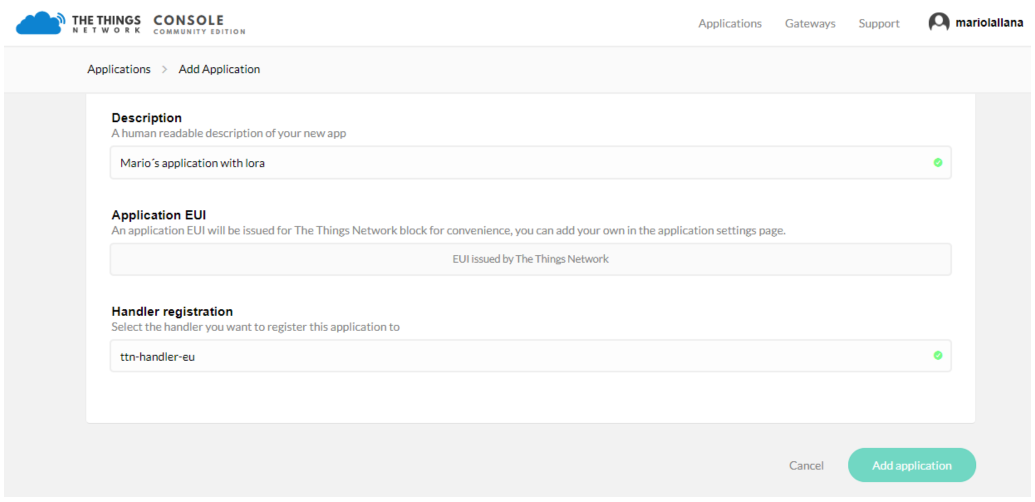
4.2.5. TheThingsNetwork application

In the TheThingNetwork webpage we will have to create two things in our account to perform the test of our LoRa WAN network:

- New Application
- New Device

-Add application:

The application that we are creating will be an interface between the end-point devices and the VGTU LoRa station direction that will receive and store all the information sent.


Applications > Add Application

Description
A human readable description of your new app
Mario´s application with lora

Application EUI
An application EUI will be issued for The Things Network block for convenience, you can add your own in the application settings page.
EUI issued by The Things Network

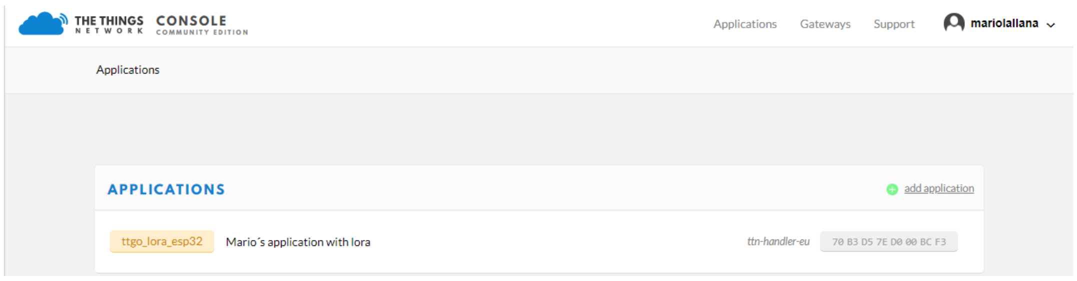
Handler registration
Select the handler you want to register this application to
ttn-handler-eu

Cancel | Add application

Figure 81 adding a TTN application

We have a group of fields that we have to fill with information. The Application ID field will be the words to identify our app, we have called it ttgo_lora_esp32 (we should put a representative name without blank spaces),The description field doesn´t affect anything to our application, its just to describe the main point of our application, The Application EUI its an important field (we will deep into it later, but we can say that the EUI it´s an identifier), but it can be filled automatically by the webpage. The Handler registration field must be filled with: ttn-handler-eu, as this is one of the possible options that the webpage give us.

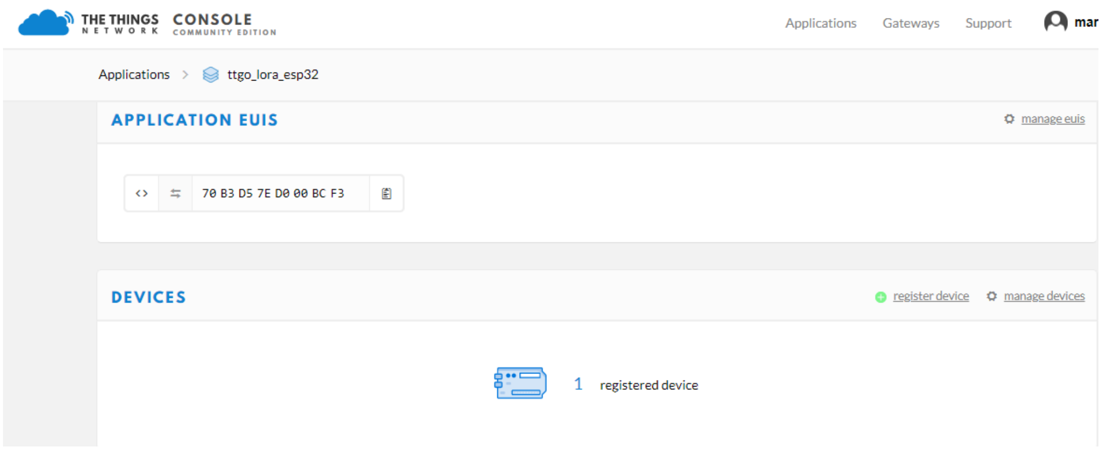
After that our application will be finished and ready to work.


APPLICATIONS — add application

ttgo_lora_esp32	Mario´s application with lora	ttn-handler-eu	70 B3 D5 7E D0 00 BC F3

Figure 82 checking our TTN app

Now if we get into our application, we wiil see the application EUIS generated by the webpage in hexadecimal format, and downer will appear our devices, so the next step will be how to add new devices to our app.



70 B3 D5 7E D0 00 BC F3

1 registered device

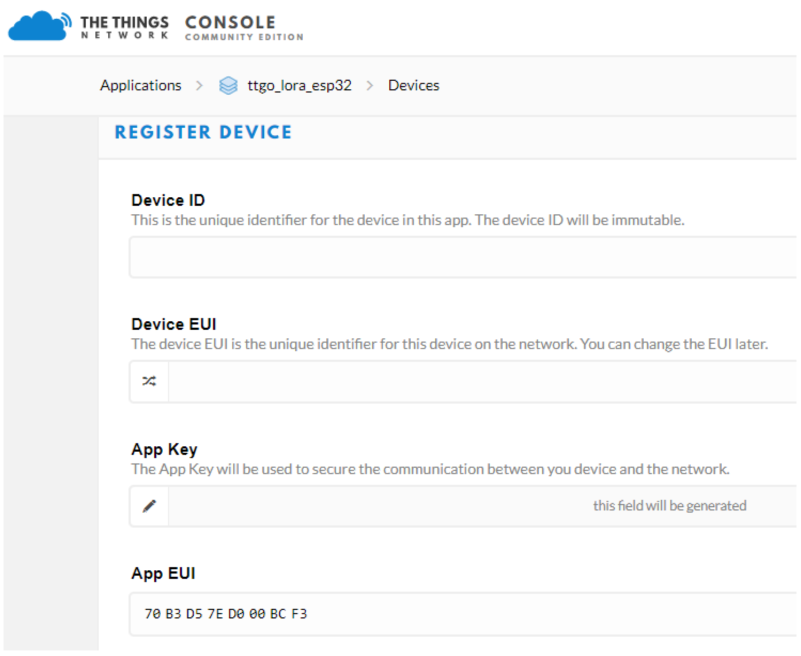
Figure 83 App EUIS and registered device in TTN

-Register Device:

We have a group of fields that we have to fill with information. The Device ID field will be the words to identify our app, we have called it ttgo_lora_esp32_v1 (we should put a representative name without blank spaces),The description field doesn´t affect anything to our device, it´s just to describe the main point of our device, The Device EUI it’s an important field (we will deep into it later, but we can say that the EUI it´s an identifier), but it can be filled automatically by the webpage with random numbers.

The App Key field will be filled automatically by the webpage with random numbers, field (we will deep into it later).

We need to put in the App EUI field the number generated for our application, to connect the app with the new device.


THE THINGS NETWORK
CONSOLE COMMUNITY EDITION
Applications > ttgo_lora_esp32 > Devices

REGISTER DEVICE

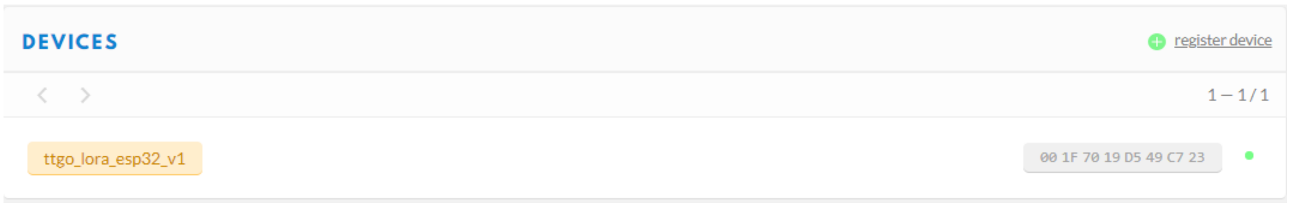
Device ID
This is the unique identifier for the device in this app. The device ID will be immutable.

Device EUI
The device EUI is the unique identifier for this device on the network. You can change the EUI later.

App Key
The App Key will be used to secure the communication between you device and the network.
this field will be generated

App EUI
70 B3 D5 7E D0 00 BC F3

Figure 84 parameters of our new TTN device


DEVICES

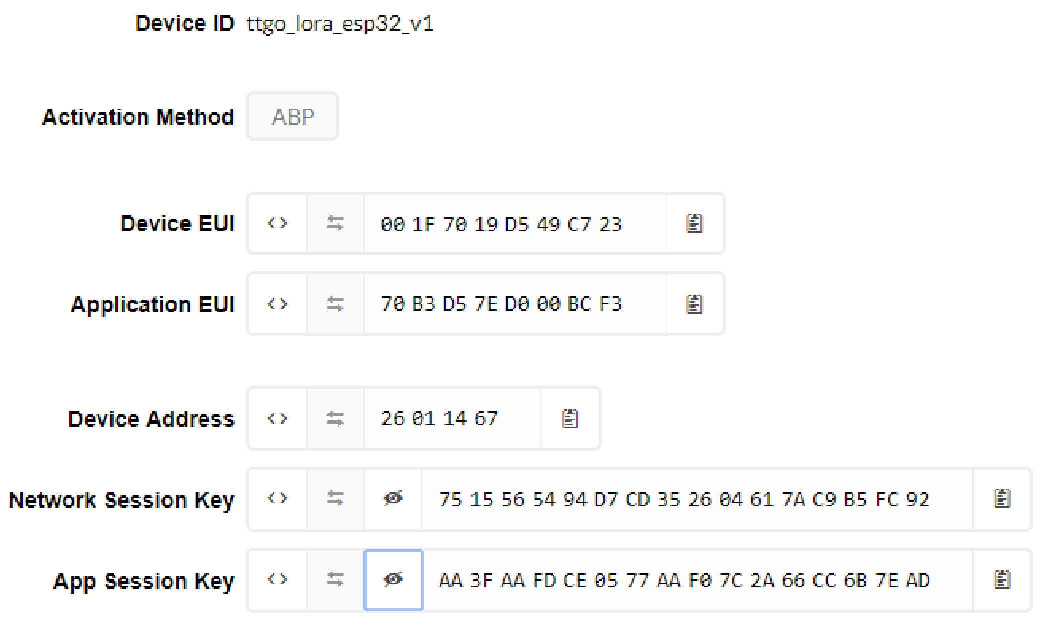
ttgo_lora_esp32_v1 00 1F 70 19 D5 49 C7 23 ●

Figure 85 status check of our TTN device

Now that we have our device registered we have to change a few parameters of its configuration.

Device Address, this field is one the most significant ones, because its filled with the "ID" of our device, the TTGO v1, in my case.

The Network Session Key, the other key field that we had fill with the "ID" of our gateway, the VGTU LoRa station.



Device ID	ttgo_lora_esp32_v1
Activation Method	ABP
Device EUI	00 1F 70 19 D5 49 C7 23
Application EUI	70 B3 D5 7E D0 00 BC F3
Device Address	26 01 14 67
Network Session Key	75 15 56 54 94 D7 CD 35 26 04 61 7A C9 B5 FC 92
App Session Key	AA 3F AA FD CE 05 77 AA F0 7C 2A 66 CC 6B 7E AD

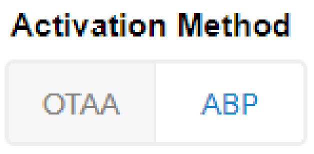
Figure 86 EUIS and Keys of our device

We change the device address, the network session key and the app session key in our own code, also we have to change the configuration of activation method to ABP [33].

4.2.6. Two more basic notions of LoRa WAN:

-End-Device setup and activation method:

You can choose for either OTAA(Over the air activation) or ABP(Activation by personalization). I chose ABP, because its easier to configure and use.



Activation Method

OTAA [✓ ABP]

Figure 87 button of TTN app to choose the activation method

We are going to take a closer look and see what do those two group of words mean. LoRaWAN allows for its packets to be both signed and encrypted by the use of keys known to both the node (module/sensor) and the LoRaWAN network/application server. The following are the keys:

- Network Session Key (NwkSKey)
- Application Session Key (AppSKey)

In addition, each device on the network has a unique device address (DevAddr).

These keys are known only to the individual node (RM1xx module) and the network/application server. This means that another node or man-in-the-middle is not able to decode the packet payload.

The following two methods are used to deploy these keys:

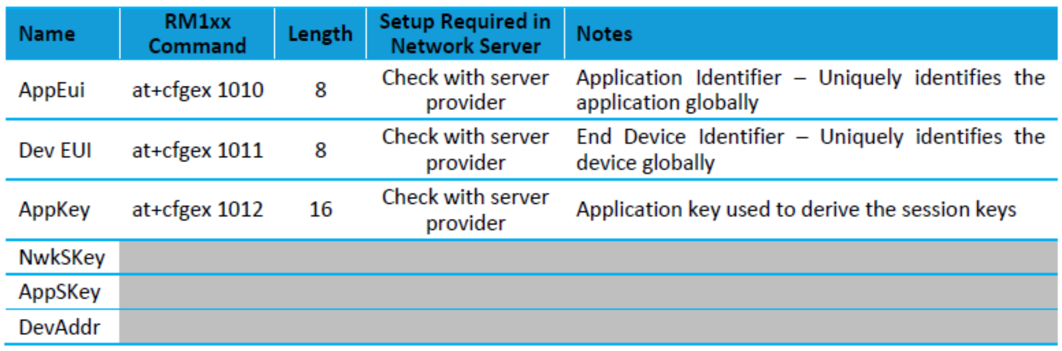
- Over-the-Air Activation (OTAA)
- Activation by personalization (ABP)

With either of these methods, the same keys are loaded into both the module and network server that allows end-to-end data security.

Over-the-Air activation:

Over-the-Air activation (OTAA) [34] uses an application ID (AppEUI) and an application key (AppKey) along with a device ID (DevEui) to derive the network session key (NwkSKey), application session key (AppSKey) and the device address. A device address (DevAddr) is assigned by the network. OTAA is the preferred method because of its security advantages including regeneration of the session keys and simplifying network management.

In conclusion, a join procedure with a join-request and a join-accept message exchange is used for each new session. Based on the join-accept message, the end-devices are able to obtain the new session keys (NwkSkey and AppSKey).



Name	RM1xx Command	Length	Setup Required in Network Server	Notes
AppEui	at+cfgex 1010	8	Check with server provider	Application Identifier – Uniquely identifies the application globally
Dev EUI	at+cfgex 1011	8	Check with server provider	End Device Identifier – Uniquely identifies the device globally
AppKey	at+cfgex 1012	16	Check with server provider	Application key used to derive the session keys
NwkSKey				
AppSKey				
DevAddr				

Table 7 parameters of the OTAA [34]

Activation by personalization:

Activation by personalization (ABP) [34] uses both session keys directly, along with the DevAddr, to sign and encrypt the data packets. These must be configured both on the node (RM1xx module) and on the network server, expanding these ideas.

The activation process should give the following information to an end-device:

- End-device address (DevAddr): A 32-bit identifier of the end-device. Seven bits are used as the network identifier, and 25 bits are used as the network address of the end-device.
- Application identifier (AppEUI): A global application ID in the IEEE EUI64 address space that uniquely identifies the owner of the end-device.
- Network session key (NwkSKey): A key used by the network server and the end-device to calculate and verify the message integrity code of all data messages to ensure data integrity.
- Application session key (AppSKey): A key used by the network server and end-device to encrypt and decrypt the payload field of data messages.
- In our particular case, for the ABP, the two session keys are directly stored into the end-devices.

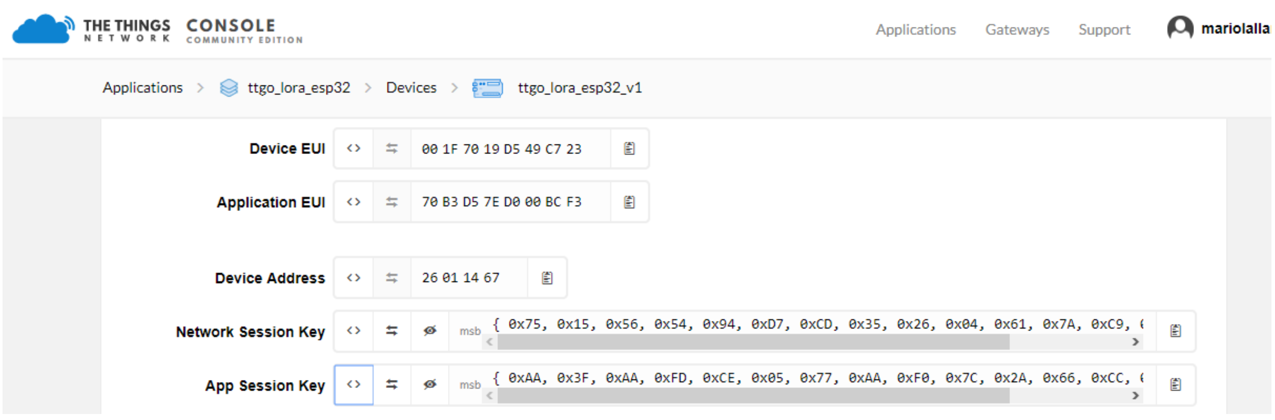
Name	RM1xx Command	Length	Setup Required in Network Server	Notes
AppEui			Check with server provider	Application Identifier – Uniquely identifies the application globally. Assigned by network server
Dev EUI			Check with server provider	End Device Identifier – Uniquely identifies the device globally. Assigned by network server
AppKey			No	Not used with ABP
NwkSKey	at+cfgex 1013	16	Yes	Network session key – Specific to the end device
AppSKey	at+cfgex 1014	16	Yes	Application session key – Specific to the end device
DevAddr	at+cfgex 1015	4	Yes	End device address – Identifies the device within the current network

Table 8 parameters of the ABP [34]

Of course you will need to create a new device in the TTN console first.

-Addressing:

Devices and application s have 64 bit unique identifier (DevEUI and AppEUI). When a device joins the network, it receives a dynamic (non-unique) 32-bit address (DevAddr)


THE THINGS NETWORK CONSOLE COMMUNITY EDITION
Applications
Gateways
Support
mariolalla
Applications > ttgo_lora_esp32 > Devices > ttgo_lora_esp32_v1

Device EUI: 00 1F 70 19 D5 49 C7 23

Application EUI: 70 B3 D5 7E D0 00 BC F3

Device Address: 26 01 14 67

Network Session Key: msb { 0x75, 0x15, 0x56, 0x54, 0x94, 0xD7, 0xCD, 0x35, 0x26, 0x04, 0x61, 0x7A, 0xC9,

App Session Key: msb { 0xAA, 0x3F, 0xAA, 0xFD, 0xCE, 0x05, 0x77, 0xAA, 0xF0, 0x7C, 0x2A, 0x66, 0xCC,

Figure 88 updating the keys in the Arduino code

For the update of the keys in the code we are using, the network session key and the app session key we have to change the values to hexadecimal style, we don´t change the device address format though.

-Additional coding lines needed the make the OLED display works properly:

```
#include <U8x8lib.h>

#define BUILTIN_LED 25

// the OLED used
```

```
        U8X8_SSD1306_128X64_NONAME_SW_I2C  u8x8(/* clock=*/ 15, /*
data=*/ 4, /* reset=*/ 16);
```

We also changed the pins of our pin connection in arduino programm fot the OLED, because as we already knowed, V1 and V2 of ttgo module are different in the pinmapping.

The three pins which will make the comunications with the I2C protocol will be 4, 15 and 16.

4.3. Final working test:

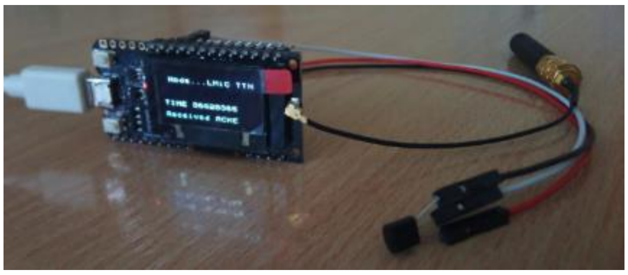
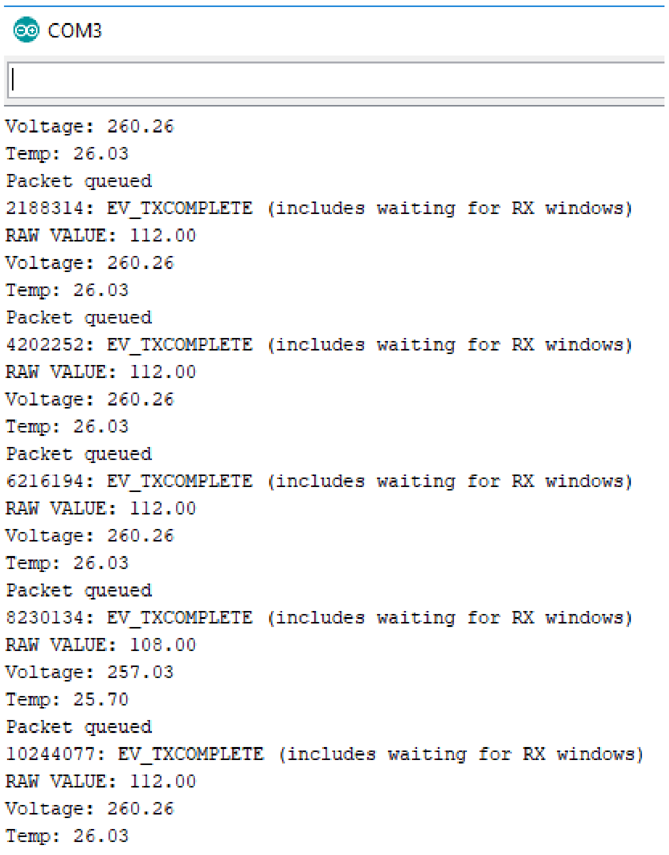


Figure 89 TTGO module powered and LM35 sensor attached to it

Now we have everything ready to work. The code will be the last shown, the test_abp_lorawan_LM35sensor2.ino, this codes implements everything we are going to need: LoRa WAN protocols, sensor reading, AD conversion , OLED display controller pin selection, LoRa transceiver pin selection, all the keys, addresses and EUIS needed, and selection of channel for transmission.

We can see in the Arduino serial monitor how the event EV_TXCOMPLETE, that means that the data on the buffer has been sent, and downstream in has been received in return perfectly. And in each line; the raw data, voltage and temperature that we have been sent is shown.



```
Voltage: 260.26
Temp: 26.03
Packet queued
2188314: EV_TXCOMPLETE (includes waiting for RX windows)
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
Packet queued
4202252: EV_TXCOMPLETE (includes waiting for RX windows)
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
Packet queued
6216194: EV_TXCOMPLETE (includes waiting for RX windows)
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
Packet queued
8230134: EV_TXCOMPLETE (includes waiting for RX windows)
RAW VALUE: 108.00
Voltage: 257.03
Temp: 25.70
Packet queued
10244077: EV_TXCOMPLETE (includes waiting for RX windows)
RAW VALUE: 112.00
Voltage: 260.26
Temp: 26.03
```

Figure 90 Arduino serial monitor

We can also see on the display when the TTGO has received the ACKE confirmation message:

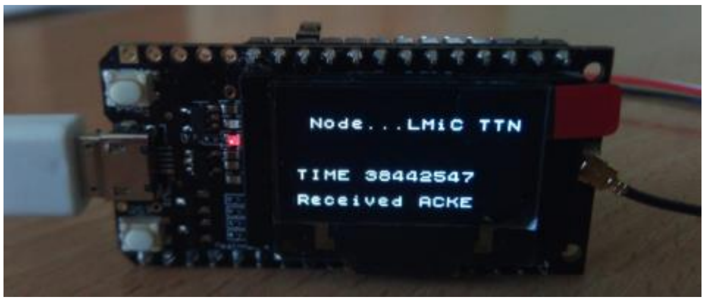
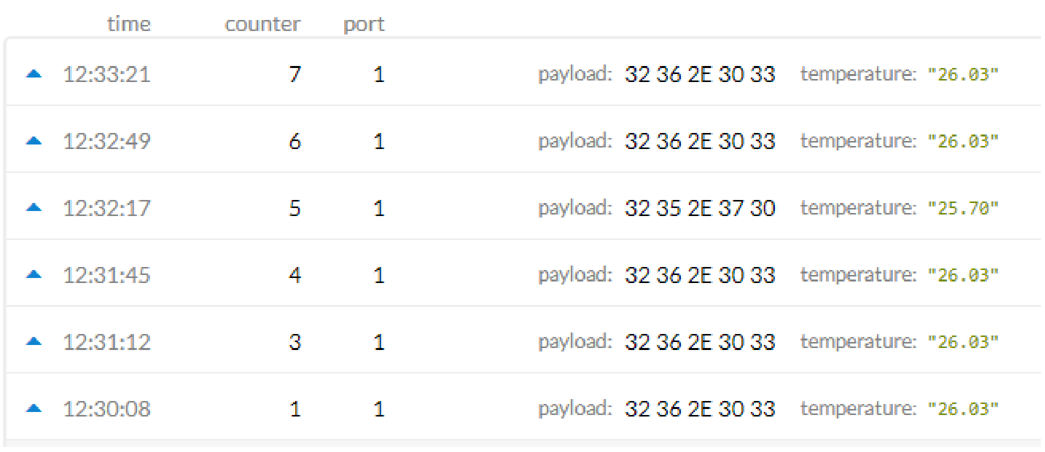


Figure 91 OLED displaying ACKE message received

To see if our project is working the it has to, let´s take a look at the TTN application: let´s check if it´s receiving data, and what data is it receiving too check if it’s the same.



	time	counter	port		
▲	12:33:21	7	1	payload: 32 36 2E 30 33	temperature: "26.03"
▲	12:32:49	6	1	payload: 32 36 2E 30 33	temperature: "26.03"
▲	12:32:17	5	1	payload: 32 35 2E 37 30	temperature: "25.70"
▲	12:31:45	4	1	payload: 32 36 2E 30 33	temperature: "26.03"
▲	12:31:12	3	1	payload: 32 36 2E 30 33	temperature: "26.03"
▲	12:30:08	1	1	payload: 32 36 2E 30 33	temperature: "26.03"

Figure 92 Data received and decoded in our TTN app.

It´s easy to conclude that:

- We are sending data
- The data we are sending is the correct and the same as the data displayed on the serial monitor.

5.CONCLUSIONS.

I would like to start saying that the Semtech LoRa technology, in this case the SX1276 transceiver, is a success in terms of its accessibility when implementing a LoRa WAN network, they have an achieved a reliable, secure and open IoT solution, accessible to almost everyone interested in that field of technology, and furthermore is not expensive to start a project with it. Also, the ESP32 seems to be a good option to be the “main brain” of IoT projects.

Finally, we have succeeded in our main goal: transmitting temperature data through LoRa technology in a LoRa network. After a great number of tests, such as the OLED display test, the bare LoRa packages sending test between two LoRa modules, the sensors tests (10K potentiometer and LM35) with different power sources, the LoRa WAN implementation with ABP sending just a constant string, finally, the most important test; the mixture of everything, the LoRa WAN implementation with ABP configuration transmitting the temperature data of the LM35 through the LoRa Gateway to our server network in the cloud of TheThingsNetwork.org.

However, all tests have brought to light a couple of failures. On the one hand, the ADC of the ESP32 has a threshold level before it senses any low voltage values, this is a problem when we are using a low voltage output sensor like the LM35, which in a regular environment, like the earth, is never going to provide an output above 500mV (50ºC). But as we have probed with a small correction in the offset, this can be fixed. So, the ESP32 may not be suitable for low power signal sensors. On the other hand, the LoRa technology seems adequate in many other aspects, but the antenna hasn’t got a strong working range, with regards to tests using the LoRa gateway located at the faculty, these had to be carried out within the faculty´s area as its hardware only covers a small region.

6.REFERENCES.

[1] Jen Clark. (n.d.). What is the Internet of Things, and how does it work? Retrieved May 31, 2018, from https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/

[2] DigiKey. (n.d.). Descripción general de la plataforma LoRa. Retrieved May 17, 2018, from https://www.digikey.com/es/articles/techzone/2017/jun/develop-lora-for-low-rate-long-range-iot-applications

[3] LoRa Architecture - LoRaWAN Tutorial. (n.d.). Retrieved May 17, 2018, from http://www.3glteinfo.com/lora/lora-architecture/

[4] Microchip. (2016). *LoRa Technology Gateway User's Guide*.

[5] LoRa CHIRP - YouTube. (n.d.). Retrieved May 31, 2018, from https://www.youtube.com/watch?v=dxYY097QNs0

[6] Augustin, A., Yi, J., Clausen, T., & Townsley, W. (2016). A Study of LoRa: Long Range &amp; Low Power Networks for the Internet of Things. *Sensors*, *16*(12), 1466. http://doi.org/10.3390/s16091466

[7] Pérez García DIRECTORES, R., Gómez Montenegro, C., Vidal Ferré, R., & Pérez García Directores, R. (n.d.). TRABAJO FINAL DE GRADO TÍTULO DEL TFG: Evaluación de LoRa/LoRaWAN para escenarios de Smart City TITULACIÓN: Grado en Ingeniería Telemática. Retrieved from https://upcommons.upc.edu/bitstream/handle/2117/100922/memoria.pdf

[8] LoRa (Long Range) End Device Classifications - Techplayon. (n.d.). Retrieved May 31, 2018, from http://www.techplayon.com/lora-long-range-end-device-classifications-class-class-b-class-c/

[9] LoRaWAN | The Things Network. (n.d.). Retrieved May 31, 2018, from https://www.thethingsnetwork.org/docs/lorawan/

[10] Heltec WiFi LoRa 32 – ESP32 with OLED and SX1278 – Robot Zero One. (n.d.). Retrieved May 31, 2018, from https://robotzero.one/heltec-wifi-lora-32/

[11] https://github.com/LilyGO/TTGO-LORA32-V2.0

[12] Temperature and humidity module DHT11 Product Manual. (n.d.). Retrieved from www.aosong.com

[13] HL-69 Soil Hygrometer Moisture Sensor | Soldering Sunday. (n.d.). Retrieved May 31, 2018, from https://solderingsunday.com/shop/arduino/hl-69-soil-hygrometer-moisture-sensor/

[14] HC-SR04 User Guide. (n.d.). Retrieved from https://www.mpja.com/download/hc-sr04_ultrasonic_module_user_guidejohn.pdf

[15] MMA7361 Sensor Acelerometro - HETPRO/TUTORIALES. (n.d.). Retrieved May 31, 2018, from https://hetpro-store.com/TUTORIALES/mma7361-sensor-acelerometro/

[16] LM35 LM35 Precision Centigrade Temperature Sensors. (n.d.). Retrieved from http://www.ti.com/lit/ds/symlink/lm35.pdf

[17] Analog to Digital Converter — ESP-IDF Programming Guide v3.0-dev-2583-g64b56be documentation. (n.d.). Retrieved May 31, 2018, from http://esp-idf.readthedocs.io/en/latest/api-reference/peripherals/adc.html

[18] Razupai magajin. 2017-12. (2017). Nikkeibipisha. Retrieved from http://esp32.net/

[19] https://github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/windows.md

[20] Systems, E. (2016). ESP32 Datasheet, 1–43. Retrieved from http://espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[21] Description, G., Features, K. E. Y. P., & Information, O. (2016). Sx1276/77/78/79, (August).

[22] Kiara Navarro. (n.d.). ¿Cómo funciona el protocolo SPI? Retrieved May 17, 2018, from http://panamahitek.com/como-funciona-el-protocolo-spi/

[23] Systech, S. (2005). SSD1306 Datasheet. *Rivers*, 1–50.

[24] Valdez, J., & Becker, J. (2015). Understanding the I 2 C Bus. Retrieved from http://www.ti.com/lit/an/slva704/slva704.pdf

[25] IBM. (2015). IBM LoRaWAN in C (LMiC) Technical Specification. *IBM Research Labs(LMiC)*, (LMiC). Retrieved from https://www.zurich.ibm.com/

[26] Adventures with Lora: Building an IoT temperature sensor. Episode 3 Combining sensor data and LoraWan. (n.d.). Retrieved May 31, 2018, from http://talk2lora.blogspot.com/2016/11/building-iot-temperature-sensor-episode_27.html

[27] CONVERTIDOR DE APROXIMACIONES SUCESIVAS. (n.d.). Retrieved from http://www.geocities.ws/pnavar2/convert/adc/adc_34.html

[28] Analog to Digital Converter — ESP-IDF Programming Guide v2.0 documentation. (n.d.). Retrieved May 31, 2018, from https://esp-idf.readthedocs.io/en/v2.0/api/peripherals/adc.html

[29] My Chinese ESP32 + SX1276 board + DHT11 connected to The Things Network – #1 – ICT en Onderwijs BLOG. (n.d.). Retrieved May 31, 2018, from https://ictoblog.nl/2018/01/13/my-chinese-esp32-sx1276-board-dht11-connected-to-the-things-network-1

[30] Tech Note 069 - Using the ESP32 ADC and some of its more advanced functions - YouTube. (n.d.). Retrieved May 31, 2018, from https://www.youtube.com/watch?v=RlKMJknsNpo

[31] LoRa ESP32 OLED Range Test &amp; Example Wemos TTGO IoT Project - YouTube. (n.d.). Retrieved May 31, 2018, from https://www.youtube.com/watch?v=2Q4O88hmjzE

[32] Como programar los ESP32 con el IDE de Arduino - YouTube. (n.d.). Retrieved May 31, 2018, from https://www.youtube.com/watch?v=4UfkIYUMP1E

[33] Specification, L. (2015). LoRaWAN TM Specification.

[34] Note, A. (2015). LoRaWAN Keys and IDs Overview, 1–6.

7.ANEXES.

7.1. Code

7.1.1. SSD1306.h library

```
/* Drawing functions */
   // Sets the color of all pixel operations
   void setColor(OLEDDISPLAY_COLOR color);

   // Draw a pixel at given position
   void setPixel(int16_t x, int16_t y);

   // Draw a line from position 0 to position 1
   void drawLine(int16_t x0, int16_t y0, int16_t x1, int16_t y1);

   // Draw the border of a rectangle at the given location
   void drawRect(int16_t x, int16_t y, int16_t width, int16_t height);

   // Fill the rectangle
   void fillRect(int16_t x, int16_t y, int16_t width, int16_t height);

   // Draw the border of a circle
   void drawCircle(int16_t x, int16_t y, int16_t radius);

   // Draw all Quadrants specified in the quads bit mask
   void drawCircleQuads(int16_t x0, int16_t y0, int16_t radius, uint8_t quads);

   // Fill circle
   void fillCircle(int16_t x, int16_t y, int16_t radius);

   // Draw a line horizontally
   void drawHorizontalLine(int16_t x, int16_t y, int16_t length);

   // Draw a lin vertically
   void drawVerticalLine(int16_t x, int16_t y, int16_t length);

   // Draws a rounded progress bar with the outer dimensions given by width and height. Progress is
   // a unsigned byte value between 0 and 100
   void drawProgressBar(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t progress);

   // Draw a bitmap in the internal image format
   void drawFastImage(int16_t x, int16_t y, int16_t width, int16_t height, const char *image);

   // Draw a XBM
   void drawXbm(int16_t x, int16_t y, int16_t width, int16_t height, const char *xbm);

   /* Text functions */

   // Draws a string at the given location
   void drawString(int16_t x, int16_t y, String text);

   // Draws a String with a maximum width at the given location.
   // If the given String is wider than the specified width
   // The text will be wrapped to the next line at a space or dash
```

```
  void drawStringMaxWidth(int16_t x, int16_t y, uint16_t maxLineWidth, String text);

  // Returns the width of the const char* with the current
  // font settings
  uint16_t getStringWidth(const char* text, uint16_t length);

  // Convencience method for the const char version
  uint16_t getStringWidth(String text);

  // Specifies relative to which anchor point
  // the text is rendered. Available constants:
  // TEXT_ALIGN_LEFT, TEXT_ALIGN_CENTER, TEXT_ALIGN_RIGHT, TEXT_ALIGN_CENTER_BOTH
  void setTextAlignment(OLEDDISPLAY_TEXT_ALIGNMENT textAlignment);

  // Sets the current font. Available default fonts
  // ArialMT_Plain_10, ArialMT_Plain_16, ArialMT_Plain_24
  void setFont(const char *fontData);

  /* Display functions */

  // Turn the display on
  void displayOn(void);

  // Turn the display offs
  void displayOff(void);

  // Inverted display mode
  void invertDisplay(void);

  // Normal display mode
  void normalDisplay(void);

  // Set display contrast
  void setContrast(char contrast);

  // Turn the display upside down
  void flipScreenVertically();
```

7.1.2. Oled_LoRa_sender.ino

```
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include "SSD1306.h"
#include "images.h"
#define SCK     5    // GPIO5  -- SX1278's SCK
#define MISO    19   // GPIO19 -- SX1278's MISO
#define MOSI    27   // GPIO27 -- SX1278's MOSI
#define SS      18   // GPIO18 -- SX1278's CS
#define RST     14   // GPIO14 -- SX1278's RESET
#define DI0     26   // GPIO26 -- SX1278's IRQ(Interrupt Request)
#define BAND    868E6

unsigned int counter = 0;

```

```
//V1 modification
SSD1306 display(0x3c, 4, 15);
String rssi = "RSSI --";
String packSize = "--";
String packet ;

void logo(){
  display.clear();
  display.drawXbm(0,5,logo_width,logo_height,logo_bits);
  display.display();
}

void setup() {
  pinMode(16,OUTPUT);
  pinMode(2,OUTPUT);

  digitalWrite(16, LOW);    // set GPIO16 low to reset OLED
  delay(50);
  digitalWrite(16, HIGH); // while OLED is running, must set GPIO16 in high

  Serial.begin(9600);
  while (!Serial);
  Serial.println();
  Serial.println("LoRa Sender Test");

  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  if (!LoRa.begin(868E6)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  //LoRa.onReceive(cbk);
//  LoRa.receive();
  Serial.println("init ok");
  display.init();
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  logo();
  delay(1500);
}

void loop() {
  display.clear();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);

  display.drawString(0, 0, "Sending packet: ");
  display.drawString(90, 0, String(counter));
  display.display();

  // send packet
  LoRa.beginPacket();
  LoRa.print("hello ");
  LoRa.print(counter);
  LoRa.endPacket();

  counter++;
  digitalWrite(2, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(2, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

7.1.3. Ttgo_abp_lorawan.ino

```

#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>
#include <U8x8lib.h>

#define BUILTIN_LED 25

// the OLED used
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(/* clock=*/ 15, /* data=*/ 4, /* reset=*/ 16);

// LoRaWAN NwkSKey, network session key
// This is the default Semtech key, which is used by the early prototype TTN
// network.
static const PROGMEM u1_t NWKSKEY[16] = { 0x75, 0x15, 0x56, 0x54, 0x94, 0xD7, 0xCD, 0x35, 0x26, 0x04, 0x61, 0x7A, 0xC9, 0xB5, 0xFC, 0x92 };

// LoRaWAN AppSKey, application session key
// This is the default Semtech key, which is used by the early prototype TTN
// network.
static const u1_t PROGMEM APPSKEY[16] = { 0xAA, 0x3F, 0xAA, 0xFD, 0xCE, 0x05, 0x77, 0xAA, 0xF0, 0x7C, 0x2A, 0x66, 0xCC, 0x6B, 0x7E, 0xAD };

// LoRaWAN end-device address (DevAddr)
static const u4_t DEVADDR = 0x26011467; // <-- Change this address for every node!

// These callbacks are only used in over-the-air activation, so they are
// left empty here (we cannot leave them out completely unless
// DISABLE_JOIN is set in config.h, otherwise the linker will complain).
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }

static uint8_t mydata[] = "Hello, world!";
static osjob_t sendjob;

// Schedule TX every this many seconds (might become longer due to duty
// cycle limitations).
const unsigned TX_INTERVAL = 30;

// Pin mapping
const lmic_pinmap lmic_pins = {
  .nss = 18,
  .rxtx = LMIC_UNUSED_PIN,
  .rst = 14,
  .dio = {26, 33, 32},
};

void onEvent (ev_t ev) {
    Serial.print(os_getTime());
    u8x8.setCursor(0, 5);
    u8x8.printf("TIME %lu", os_getTime());
    Serial.print(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            Serial.println(F("EV_SCAN_TIMEOUT"));
            u8x8.drawString(0, 7, "EV_SCAN_TIMEOUT");
            break;
        case EV_BEACON_FOUND:
            Serial.println(F("EV_BEACON_FOUND"));
            u8x8.drawString(0, 7, "EV_BEACON_FOUND");
            break;
        case EV_BEACON_MISSED:
            Serial.println(F("EV_BEACON_MISSED"));
            u8x8.drawString(0, 7, "EV_BEACON_MISSED");
            break;
        case EV_BEACON_TRACKED:
            Serial.println(F("EV_BEACON_TRACKED"));
            u8x8.drawString(0, 7, "EV_BEACON_TRACKED");
            break;
        case EV_JOINING:
            Serial.println(F("EV_JOINING"));
            u8x8.drawString(0, 7, "EV_JOINING");
            break;
        case EV_JOINED:
            Serial.println(F("EV_JOINED"));
            u8x8.drawString(0, 7, "EV_JOINED ");
            // Disable link check validation (automatically enabled
            // during join, but not supported by TTN at this time).
            LMIC_setLinkCheckMode(0);
            break;
        case EV_RFU1:
            Serial.println(F("EV_RFU1"));
            u8x8.drawString(0, 7, "EV_RFUI");
            break;
        case EV_JOIN_FAILED:
            Serial.println(F("EV_JOIN_FAILED"));
            u8x8.drawString(0, 7, "EV_JOIN_FAILED");
            break;
        case EV_REJOIN_FAILED:
            Serial.println(F("EV_REJOIN_FAILED"));
            u8x8.drawString(0, 7, "EV_REJOIN_FAILED");
            break;
        case EV_TXCOMPLETE:
            Serial.println(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            u8x8.drawString(0, 7, "EV_TXCOMPLETE");
            digitalWrite(BUILTIN_LED, LOW);
            if (LMIC.txrxFlags & TXRX_ACK)
              Serial.println(F("Received ack"));
              u8x8.drawString(0, 7, "Received ACK");
            if (LMIC.dataLen) {
                     Serial.println(F("Received "));
                     u8x8.drawString(0, 6, "RX ");
                     Serial.println(LMIC.dataLen);
                     u8x8.setCursor(4, 6);
                     u8x8.printf("%i bytes", LMIC.dataLen);
                     Serial.println(F(" bytes of payload"));
                     u8x8.setCursor(0, 7);
                     u8x8.printf("RSSI %d SNR %.1d", LMIC.rssi, LMIC.snr);
                   }
                   // Schedule next transmission
                   os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_send);
                   break;
               case EV_LOST_TSYNC:
                   Serial.println(F("EV_LOST_TSYNC"));
                   u8x8.drawString(0, 7, "EV_LOST_TSYNC");
                   break;
               case EV_RESET:
```

```
            Serial.println(F("EV_RESET"));
            u8x8.drawString(0, 7, "EV_RESET");
            break;
        case EV_RXCOMPLETE:
            // data received in ping slot
            Serial.println(F("EV_RXCOMPLETE"));
            u8x8.drawString(0, 7, "EV_RXCOMPLETE");
            break;
        case EV_LINK_DEAD:
            Serial.println(F("EV_LINK_DEAD"));
            u8x8.drawString(0, 7, "EV_LINK_DEAD");
            break;
        case EV_LINK_ALIVE:
            Serial.println(F("EV_LINK_ALIVE"));
            u8x8.drawString(0, 7, "EV_LINK_ALIVE");
            break;
         default:
            Serial.println(F("Unknown event"));
            u8x8.setCursor(0, 7);
            u8x8.printf("UNKNOWN EVENT %d", ev);
            break;
    }
}

void do_send(osjob_t* j){
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        Serial.println(F("OP_TXRXPEND, not sending"));
        u8x8.drawString(0, 7, "OP_TXRXPEND, not sent");
    } else {
        // Prepare upstream data transmission at the next possible time.
        LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);
        Serial.println(F("Packet queued"));
        u8x8.drawString(0, 7, "PACKET QUEUED");
    }
    // Next TX is scheduled after TX_COMPLETE event.
}

void setup() {
    Serial.begin(115200);
    Serial.println(F("Starting"));

    u8x8.begin();
    u8x8.setFont(u8x8_font_chroma48medium8_r);
    u8x8.drawString(0, 1, "LoRaWAN LMiC TTN Node...");

    SPI.begin(5, 19, 27);

    // LMIC init
    os_init();
    // Reset the MAC state. Session and pending data transfers will be discarded.
    LMIC_reset();

    // Set static session parameters. Instead of dynamically establishing a session
    // by joining the network, precomputed session parameters are be provided.
    #ifdef PROGMEM
    // On AVR, these values are stored in flash and only copied to RAM

    // once. Copy them to a temporary buffer here, LMIC_setSession will
```

```
            // copy them into a buffer of its own again.
            uint8_t appskey[sizeof(APPSKEY)];
            uint8_t nwkskey[sizeof(NWKSKEY)];
            memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
            memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
            LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
            #else
            // If not running an AVR with PROGMEM, just use the arrays directly
            LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
            #endif

            #if defined(CFG_eu868)
            // Set up the channels used by the Things Network, which corresponds
            // to the defaults of most gateways. Without this, only three base
            // channels from the LoRaWAN specification are used, which certainly
            // works, so it is good for debugging, but can overload those
            // frequencies, so be sure to configure the full frequency range of
            // your network here (unless your network autoconfigures them).
            // Setting up channels should happen after LMIC_setSession, as that
            // configures the minimal channel set.
            // NA-US channels 0-71 are configured automatically
        //    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        //    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);      // g-band
        //    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        //    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        //    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        //    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        //    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
            LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        //    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);      // g2-band
        //    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
            // devices' ping slots. LMIC does not have an easy way to define set this
            // frequency and support for class B is spotty and untested, so this
            // frequency is not configured here.
            #elif defined(CFG_us915)
            // NA-US channels 0-71 are configured automatically
            // but only one group of 8 should (a subband) should be active
            // TTN recommends the second sub band, 1 in a zero based count.
            // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.json
            LMIC_selectSubBand(1);
            #endif

            // Disable link check validation
            LMIC_setLinkCheckMode(0);

            // TTN uses SF9 for its RX2 window.
            LMIC.dn2Dr = DR_SF9;

            // Set data rate and transmit power for uplink (note: txpow seems t
 o be ignored by the library)
            LMIC_setDrTxpow(DR_SF7,14);

            // Start job
            do_send(&sendjob);
        }

        void loop() {
            os_runloop_once();
        }
```

7.1.4. Ttgo_abp_lorawan_LM35sensor2.ino

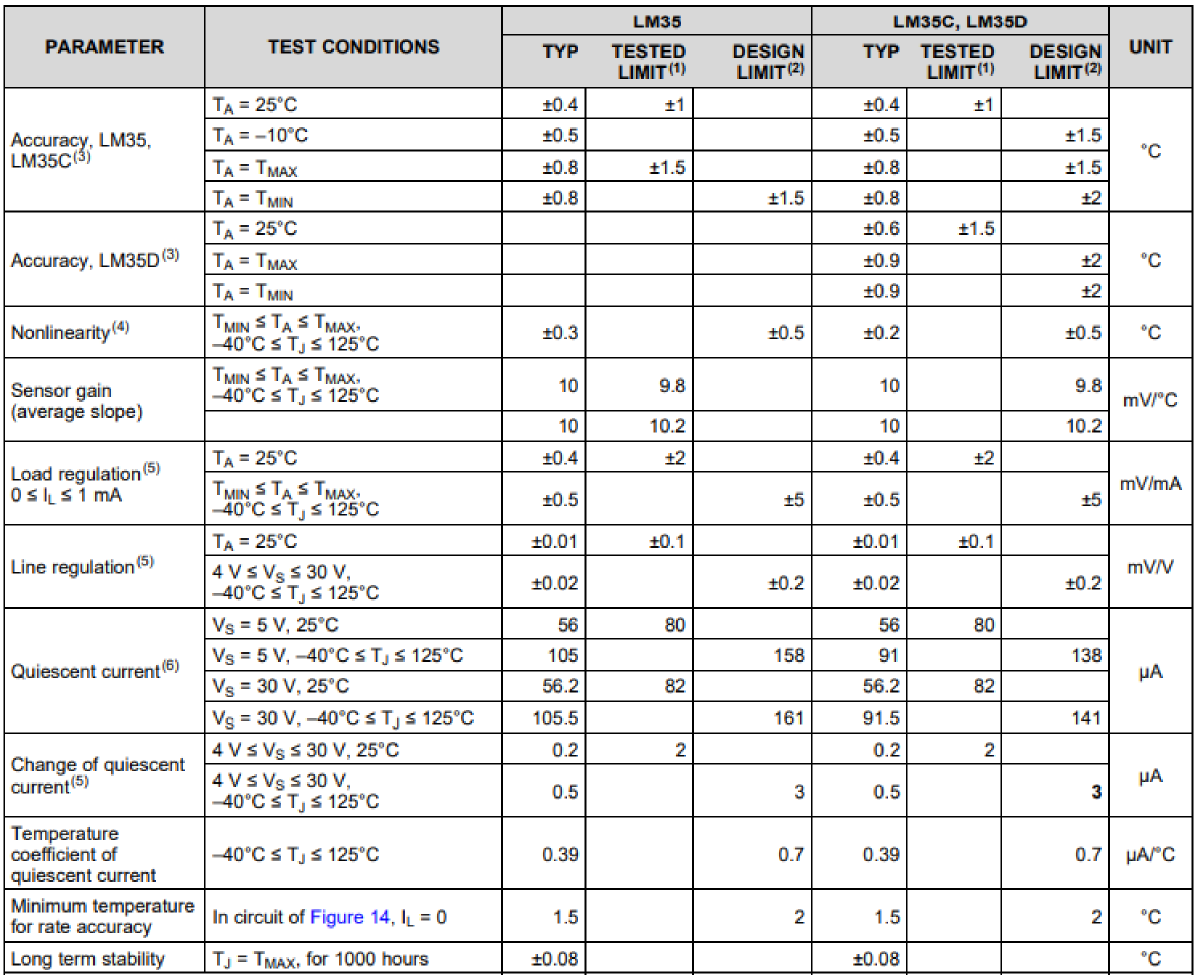
```
#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>
#include <U8x8lib.h>

#define BUILTIN_LED 25

// the OLED used
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(/* clock=*/ 15, /* data=*/ 4, /* reset=
*/ 16);

// LoRaWAN NwkSKey, network session key
// This is the default Semtech key, which is used by the early prototype TTN
// network.
static const PROGMEM u1_t NWKSKEY[16] = { 0x75, 0x15, 0x56, 0x54, 0x94, 0xD7,
0xCD, 0x35, 0x26, 0x04, 0x61, 0x7A, 0xC9, 0xB5, 0xFC, 0x92 };

// LoRaWAN AppSKey, application session key
// This is the default Semtech key, which is used by the early prototype TTN
// network.
static const u1_t PROGMEM APPSKEY[16] = { 0xAA, 0x3F, 0xAA, 0xFD, 0xCE, 0x05,
0x77, 0xAA, 0xF0, 0x7C, 0x2A, 0x66, 0xCC, 0x6B, 0x7E, 0xAD };

// LoRaWAN end-device address (DevAddr)
static const u4_t DEVADDR = 0x26011467; // <-
- Change this address for every node!

// These callbacks are only used in over-the-air activation, so they are
// left empty here (we cannot leave them out completely unless
// DISABLE_JOIN is set in config.h, otherwise the linker will complain).
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }

//static uint8_t mydata[] = "Hello, world!";
//static uint8_t mydata[] = "Hello, world!";

static uint8_t mydata[] = { 0,0,0,0,0,0,0,0};

//uint16_t mydata;
static osjob_t sendjob;

// Schedule TX every this many seconds (might become longer due to duty
// cycle limitations).
const unsigned TX_INTERVAL = 30;

// Pin mapping
const lmic_pinmap lmic_pins = {
  .nss = 18,
  .rxtx = LMIC_UNUSED_PIN,
  .rst = 14,
  .dio = {26, 33, 32},
};

void onEvent (ev_t ev) {
    Serial.print(os_getTime());
    u8x8.setCursor(0, 5);
    u8x8.printf("TIME %lu", os_getTime());
    Serial.print(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            Serial.println(F("EV_SCAN_TIMEOUT"));
            u8x8.drawString(0, 7, "EV_SCAN_TIMEOUT");
            break;
        case EV_BEACON_FOUND:
            Serial.println(F("EV_BEACON_FOUND"));
            u8x8.drawString(0, 7, "EV_BEACON_FOUND");
            break;
        case EV_BEACON_MISSED:
            Serial.println(F("EV_BEACON_MISSED"));
            u8x8.drawString(0, 7, "EV_BEACON_MISSED");
            break;
        case EV_BEACON_TRACKED:
            Serial.println(F("EV_BEACON_TRACKED"));
            u8x8.drawString(0, 7, "EV_BEACON_TRACKED");
            break;
        case EV_JOINING:
            Serial.println(F("EV_JOINING"));
            u8x8.drawString(0, 7, "EV_JOINING");
            break;
        case EV_JOINED:
            Serial.println(F("EV_JOINED"));
            u8x8.drawString(0, 7, "EV_JOINED ");
            // Disable link check validation (automatically enabled
            // during join, but not supported by TTN at this time).
            LMIC_setLinkCheckMode(0);
            break;
        case EV_RFU1:
            Serial.println(F("EV_RFU1"));
            u8x8.drawString(0, 7, "EV_RFUI");
            break;
        case EV_JOIN_FAILED:
            Serial.println(F("EV_JOIN_FAILED"));
            u8x8.drawString(0, 7, "EV_JOIN_FAILED");
            break;
        case EV_REJOIN_FAILED:
            Serial.println(F("EV_REJOIN_FAILED"));
            u8x8.drawString(0, 7, "EV_REJOIN_FAILED");
            break;
        case EV_TXCOMPLETE:
            Serial.println(F("EV_TXCOMPLETE (includes waiting for RX windows)"
));
            u8x8.drawString(0, 7, "EV_TXCOMPLETE");
            digitalWrite(BUILTIN_LED, LOW);
                    if (LMIC.txrxFlags & TXRX_ACK)
                      Serial.println(F("Received ack"));
                      u8x8.drawString(0, 7, "Received ACK");
```

```
                       if (LMIC.dataLen) {
                         Serial.println(F("Received "));
                         u8x8.drawString(0, 6, "RX ");
                         Serial.println(LMIC.dataLen);
                         u8x8.setCursor(4, 6);
                         u8x8.printf("%i bytes", LMIC.dataLen);
                         Serial.println(F(" bytes of payload"));
                         u8x8.setCursor(0, 7);
                         u8x8.printf("RSSI %d SNR %.1d", LMIC.rssi, LMIC.snr);
                       }
                       // Schedule next transmission
                       os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_I
NTERVAL), do_send);
                       break;
                   case EV_LOST_TSYNC:
                       Serial.println(F("EV_LOST_TSYNC"));
                       u8x8.drawString(0, 7, "EV_LOST_TSYNC");
                       break;
                   case EV_RESET:
                       Serial.println(F("EV_RESET"));
                       u8x8.drawString(0, 7, "EV_RESET");
                       break;
                   case EV_RXCOMPLETE:
                       // data received in ping slot
                       Serial.println(F("EV_RXCOMPLETE"));
                       u8x8.drawString(0, 7, "EV_RXCOMPLETE");
                       break;
                   case EV_LINK_DEAD:
                       Serial.println(F("EV_LINK_DEAD"));
                       u8x8.drawString(0, 7, "EV_LINK_DEAD");
                       break;
                   case EV_LINK_ALIVE:
                       Serial.println(F("EV_LINK_ALIVE"));
                       u8x8.drawString(0, 7, "EV_LINK_ALIVE");
                       break;
                    default:
                       Serial.println(F("Unknown event"));
                       u8x8.setCursor(0, 7);
                       u8x8.printf("UNKNOWN EVENT %d", ev);
                       break;
               }
           }

           void do_send(osjob_t* j){

               float temp1;
               int raw = analogRead(35);
               temp1 = raw;
               float voltag = ((temp1*3300)/4095)+170;
               float temp = (voltag*0.1);

               Serial.print("RAW VALUE: ");
               Serial.println(temp1);
               Serial.print("Voltage: ");
               Serial.println(voltag);
               Serial.print("Temp: ");
               Serial.println(temp);

               dtostrf(temp,5,2,(char*)mydata);

               // Check if there is not a current TX/RX job running
               if (LMIC.opmode & OP_TXRXPEND) {
                   Serial.println(F("OP_TXRXPEND, not sending"));
                   u8x8.drawString(0, 7, "OP_TXRXPEND, not sent");
```

```
  } else {
    // Prepare upstream data transmission at the next possible time.

    //LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);
     LMIC_setTxData2(1, mydata, strlen((char*) mydata), 0);
  }
    Serial.println(F("Packet queued"));
    u8x8.drawString(0, 7, "PACKET QUEUED");
  }
  // Next TX is scheduled after TX_COMPLETE event.


void setup() {
  Serial.begin(115200);
  Serial.println(F("Starting"));
  analogReadResolution(12);

  u8x8.begin();
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.drawString(0, 1, "LoRaWAN LMiC TTN Node...");

  SPI.begin(5, 19, 27);

  // LMIC init
  os_init();
  // Reset the MAC state. Session and pending data transfers will be discarded.
  LMIC_reset();

  // Set static session parameters. Instead of dynamically establishing a session
  // by joining the network, precomputed session parameters are be provided.
  #ifdef PROGMEM
  // On AVR, these values are stored in flash and only copied to RAM
  // once. Copy them to a temporary buffer here, LMIC_setSession will
  // copy them into a buffer of its own again.
  uint8_t appskey[sizeof(APPSKEY)];
  uint8_t nwkskey[sizeof(NWKSKEY)];
  memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
  memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
  LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
  #else
  // If not running an AVR with PROGMEM, just use the arrays directly
  LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
  #endif

  #if defined(CFG_eu868)
  // Set up the channels used by the Things Network, which corresponds
  // to the defaults of most gateways. Without this, only three base
  // channels from the LoRaWAN specification are used, which certainly
  // works, so it is good for debugging, but can overload those
  // frequencies, so be sure to configure the full frequency range of
  // your network here (unless your network autoconfigures them).
  // Setting up channels should happen after LMIC_setSession, as that
```

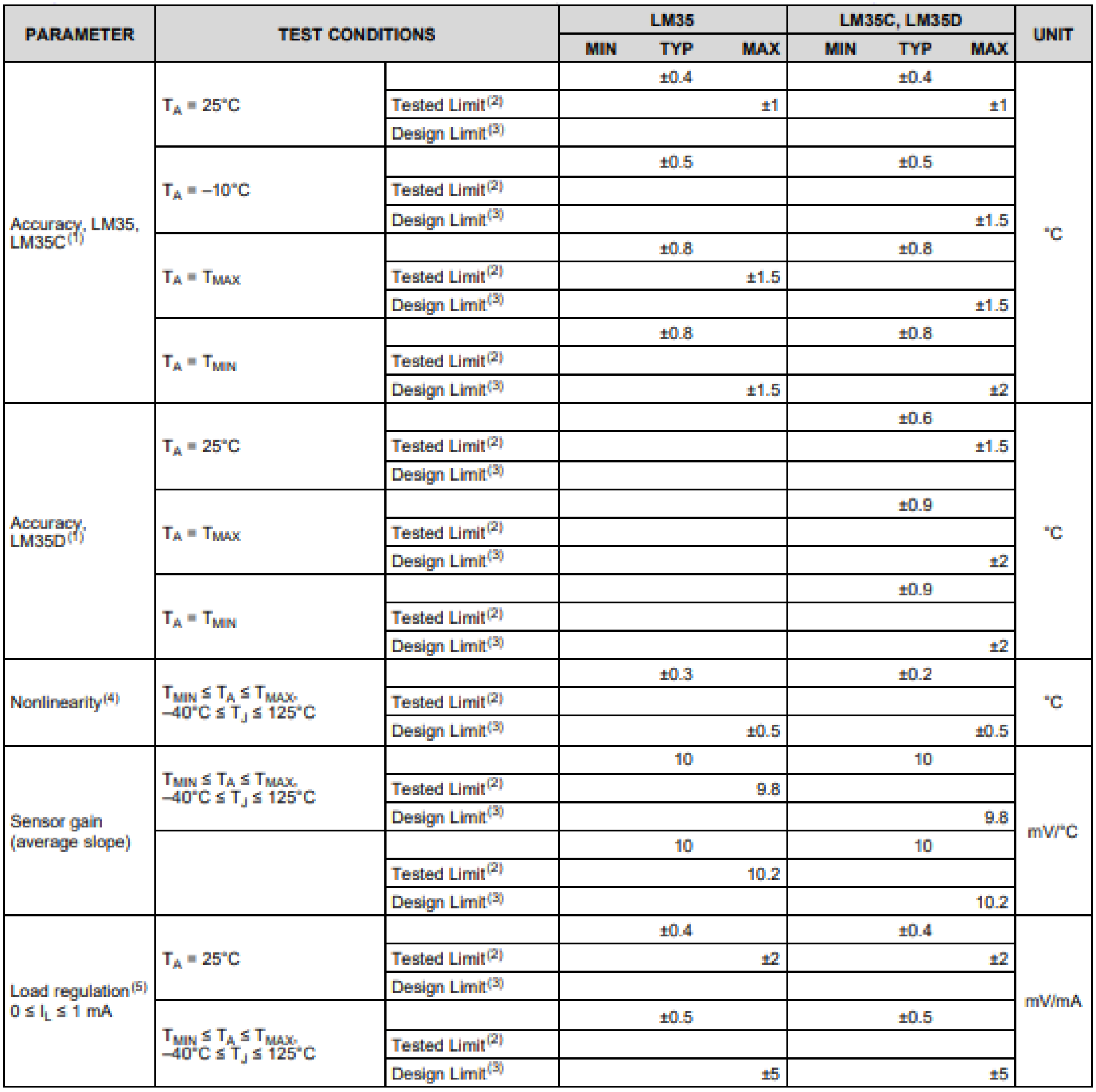
```
        // configures the minimal channel set.
        // NA-US channels 0-71 are configured automatically
    //    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
    //    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);      // g-band
    //    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
    //    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
    //    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
    //    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
    //    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
        LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);      // g-band
    //    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);      // g2-band
    //    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
        // devices' ping slots. LMIC does not have an easy way to define set this
        // frequency and support for class B is spotty and untested, so this
        // frequency is not configured here.
        #elif defined(CFG_us915)
        // NA-US channels 0-71 are configured automatically
        // but only one group of 8 should (a subband) should be active
        // TTN recommends the second sub band, 1 in a zero based count.
        // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.json
        LMIC_selectSubBand(1);
        #endif

        // Disable link check validation
        LMIC_setLinkCheckMode(0);

        // TTN uses SF9 for its RX2 window.
        LMIC.dn2Dr = DR_SF9;

        // Set data rate and transmit power for uplink (note: txpow seems to be ignored by the library)
        LMIC_setDrTxpow(DR_SF7,14);

        // Start job
        do_send(&sendjob);
    }

    void loop() {
        os_runloop_once();
    }
```

7.2. LM35 electrical characteristics



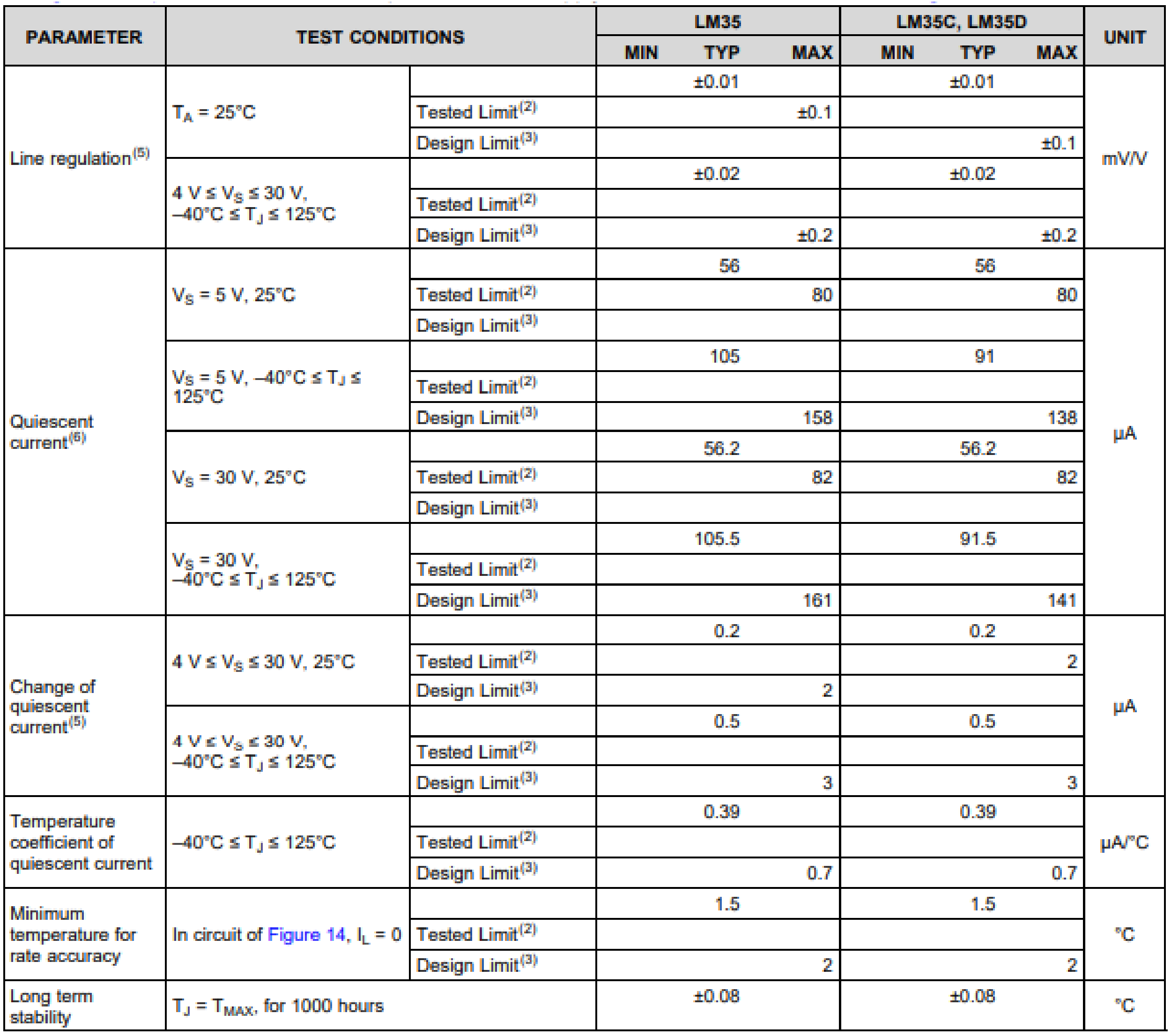
PARAMETER	TEST CONDITIONS	LM35			LM35C, LM35D			UNIT
		TYP	TESTED LIMIT (1)	DESIGN LIMIT (2)	TYP	TESTED LIMIT (1)	DESIGN LIMIT (2)	
Accuracy, LM35, LM35C (3)	T_A = 25°C	±0.4	±1		±0.4	±1		°C
	T_A = –10°C	±0.5			±0.5		±1.5	
	$T_A = T_{MAX}$	±0.8	±1.5		±0.8		±1.5	
	$T_A = T_{MIN}$	±0.8		±1.5	±0.8		±2	
Accuracy, LM35D (3)	T_A = 25°C				±0.6	±1.5		°C
	$T_A = T_{MAX}$				±0.9		±2	
	$T_A = T_{MIN}$				±0.9		±2	
Nonlinearity (4)	$T_{MIN} \le T_A \le T_{MAX}$, –40°C ≤ T_J ≤ 125°C	±0.3		±0.5	±0.2		±0.5	°C
Sensor gain (average slope)	$T_{MIN} \le T_A \le T_{MAX}$, –40°C ≤ T_J ≤ 125°C	10	9.8		10		9.8	mV/°C
		10	10.2		10		10.2	
Load regulation (5) 0 ≤ I_L ≤ 1 mA	T_A = 25°C	±0.4	±2		±0.4	±2		mV/mA
	$T_{MIN} \le T_A \le T_{MAX}$, –40°C ≤ T_J ≤ 125°C	±0.5		±5	±0.5		±5	
Line regulation (5)	T_A = 25°C	±0.01	±0.1		±0.01	±0.1		mV/V
	4 V ≤ V_S ≤ 30 V, –40°C ≤ T_J ≤ 125°C	±0.02		±0.2	±0.02		±0.2	
Quiescent current (6)	V_S = 5 V, 25°C	56	80		56	80		µA
	V_S = 5 V, –40°C ≤ T_J ≤ 125°C	105		158	91		138	
	V_S = 30 V, 25°C	56.2	82		56.2	82		
	V_S = 30 V, –40°C ≤ T_J ≤ 125°C	105.5		161	91.5		141	
Change of quiescent current (5)	4 V ≤ V_S ≤ 30 V, 25°C	0.2	2		0.2	2		µA
	4 V ≤ V_S ≤ 30 V, –40°C ≤ T_J ≤ 125°C	0.5		3	0.5		**3**	
Temperature coefficient of quiescent current	–40°C ≤ T_J ≤ 125°C	0.39		0.7	0.39		0.7	µA/°C
Minimum temperature for rate accuracy	In circuit of Figure 14, I_L = 0	1.5		2	1.5		2	°C
Long term stability	$T_J = T_{MAX}$, for 1000 hours	±0.08			±0.08			°C

Table 9 LM35 electrical characteristics part 1 [16]



PARAMETER	TEST CONDITIONS		LM35 MIN	LM35 TYP	LM35 MAX	LM35C, LM35D MIN	LM35C, LM35D TYP	LM35C, LM35D MAX	UNIT
Accuracy, LM35, LM35C[1]	$T_A = 25°C$			±0.4			±0.4		°C
		Tested Limit[2]			±1			±1	
		Design Limit[3]							
	$T_A = -10°C$			±0.5			±0.5		
		Tested Limit[2]							
		Design Limit[3]						±1.5	
	$T_A = T_{MAX}$			±0.8			±0.8		
		Tested Limit[2]			±1.5				
		Design Limit[3]						±1.5	
	$T_A = T_{MIN}$			±0.8			±0.8		
		Tested Limit[2]							
		Design Limit[3]			±1.5			±2	
Accuracy, LM35D[1]	$T_A = 25°C$						±0.6		°C
		Tested Limit[2]						±1.5	
		Design Limit[3]							
	$T_A = T_{MAX}$						±0.9		
		Tested Limit[2]							
		Design Limit[3]						±2	
	$T_A = T_{MIN}$						±0.9		
		Tested Limit[2]							
		Design Limit[3]						±2	
Nonlinearity[4]	$T_{MIN} \le T_A \le T_{MAX}$, $-40°C \le T_J \le 125°C$			±0.3			±0.2		°C
		Tested Limit[2]							
		Design Limit[3]			±0.5			±0.5	
Sensor gain (average slope)	$T_{MIN} \le T_A \le T_{MAX}$, $-40°C \le T_J \le 125°C$			10			10		mV/°C
		Tested Limit[2]			9.8				
		Design Limit[3]						9.8	
				10			10		
		Tested Limit[2]			10.2				
		Design Limit[3]						10.2	
Load regulation[5] $0 \le I_L \le 1$ mA	$T_A = 25°C$			±0.4			±0.4		mV/mA
		Tested Limit[2]			±2			±2	
		Design Limit[3]							
	$T_{MIN} \le T_A \le T_{MAX}$, $-40°C \le T_J \le 125°C$			±0.5			±0.5		
		Tested Limit[2]							
		Design Limit[3]			±5			±5	

Table 10 LM35 electrical characteristics part 2 [16]



PARAMETER	TEST CONDITIONS		LM35			LM35C, LM35D			UNIT
			MIN	TYP	MAX	MIN	TYP	MAX	
Line regulation[5]	T_A = 25°C			±0.01			±0.01		mV/V
		Tested Limit[2]			±0.1				
		Design Limit[3]						±0.1	
	4 V ≤ V_S ≤ 30 V, –40°C ≤ T_J ≤ 125°C			±0.02			±0.02		
		Tested Limit[2]							
		Design Limit[3]			±0.2			±0.2	
Quiescent current[6]	V_S = 5 V, 25°C			56			56		µA
		Tested Limit[2]			80			80	
		Design Limit[3]							
	V_S = 5 V, –40°C ≤ T_J ≤ 125°C			105			91		
		Tested Limit[2]							
		Design Limit[3]			158			138	
	V_S = 30 V, 25°C			56.2			56.2		
		Tested Limit[2]			82			82	
		Design Limit[3]							
	V_S = 30 V, –40°C ≤ T_J ≤ 125°C			105.5			91.5		
		Tested Limit[2]							
		Design Limit[3]			161			141	
Change of quiescent current[5]	4 V ≤ V_S ≤ 30 V, 25°C			0.2			0.2		µA
		Tested Limit[2]						2	
		Design Limit[3]			2				
	4 V ≤ V_S ≤ 30 V, –40°C ≤ T_J ≤ 125°C			0.5			0.5		
		Tested Limit[2]							
		Design Limit[3]			3			3	
Temperature coefficient of quiescent current	–40°C ≤ T_J ≤ 125°C			0.39			0.39		µA/°C
		Tested Limit[2]							
		Design Limit[3]			0.7			0.7	
Minimum temperature for rate accuracy	In circuit of Figure 14, I_L = 0			1.5			1.5		°C
		Tested Limit[2]							
		Design Limit[3]			2			2	
Long term stability	T_J = T_{MAX}, for 1000 hours			±0.08			±0.08		°C

Table 11 LM35 electrical characteristics part 3 [16]

7.3. Table of potentiometer test measurements:

Vexp (mV)	VADC (mV)	Vexp (mV)	VADC (mV)	Vexp (mV)	VADC (mV)
7,9	0	260,1	92	804	610
21,2	0	270	108	840	645
37	0	298,2	132	879	683
47	0	331,7	168	916	722
65	0	347,7	178	957	756,73
81,9	0	392,4	220	973	774
96,8	0	427,8	257	999	796
108,3	0	484	305	1022	823
121,2	0	508	325	1127	928,35
137	0	543	361,03	1214	1007,33
152,9	0	582	399,71	1317	1110
163,2	25	601	422,27	1414	1201
174,4	35	637	456,12	1500	1286,92
197,2	40	659	477,07	1615	1397
207,4	50	685	500	1705	1484,7
232	75	719	530	1831	1600
247,3	85	745	556	1956	1720
260,1	92	771	581,22	2071	1928,08

Table 12 Values of potentiometer test measurements, V measured in front of V provided by the ADC