Pico-Grid with LoRa 2.4 GHz

by

Kenneth Mallabo

This report is submitted in partial fulfilment of the requirements of the BSc Honours Degree in Networking Applications & Services, of the Technological University Dublin

May 22nd, 2023

Supervisor: **Frank Duignan**

School of Electrical and Electronic Engineering

Programme Chair:
Dr Anthony Grennan

DECLARATION

I, the undersigned, declare that this report is entirely my own written work, except where otherwise accredited, and that it has not been submitted for a degree or award to any other university or institution.

Signed: KMallabo

Date: **22/05/2023**

Contents

Table of Figures

Abstract

Renewable energy such as wind and solar are inherently unpredictable which may cause brownouts or blackouts in an electric grid. This document attempts to create a “Pico-Grid” which is a type of smart grid that is on a small scale, between different floors within the same building in the hopes it may alleviate the aforementioned problem.

In this project instead of using a real smart metre or inverter, it will only create a simulation of such devices but only with dummy data. Its focus will mainly be on the communication side using a variety of communication protocols and software rather than physical energy exchange.

Keywords: smart grids, Pico-Grid, Arduino Nano, WiMODino, LoRa 2.4 GHz, ChirpStack, InfluxDB, Modbus.

Chapter 1 Introduction

1.1 Introduction

In this final year project, the goal is to create a “Pico-Grid”. In contrast to a microgrid or a community grid, a Pico-Grid would be created using a simulation of integrated smart batteries and grid-tie storage inverters that would exchange energy between different floors in a building. The floors would be able to export and import energy between them. This would require an IoT solution to connect the components in a bidirectional communication system that is reasonably secure, and reliable and high throughputs are desired. With this setup there would be no need to worry about enforcing disturbance-neutrality as excess energy will not spill into the electrical grid, thus there would also be no need to worry about Ireland’s Distribution System Operator (DSO).

The project can be divided into three main stages:

- LoRa 2.4 GHz Network - Establish communication between the EMB-Fem2GW-2G4-O gateway to the WiMODino. If procuring the EMB-Fem2GW-2G4-O gateway is not possible, then using the SX1280Z3DSFGW1 would be a suitable alternative. The WiMODino has already been programmed to communicate with a LoRa 2.4 GHz gateway and would require small modifications to the configurations to change the gateway to connect to. There would be changes needed if using the SX1280Z3DSFGW1. The EMB-Fem2GW-2G4-O is an out-of-the-box device with a preinstalled OS and configuration page that would hasten the connection process. The network server to enable LoRaWAN should be ChirpStack as the network server can be implemented into the gateway itself instead of using a cloud-based service such as The Things Network.
- Pico-Grid - Mimic the predicted behaviour and functionalities of inverters, smart batteries and smart meters into the WiMODino. This would require reading into the documentation on the various devices and working with an expert on this particular subject. The WiMODino would then need to be programmed to transmit data from devices. The gateway would also need to transmit data, especially to the “inverter” to issue commands such as exporting energy.
- Energy Trading Platform - Create a simple simulated energy trading platform. What would be required is a database to store the data of a prosumer, a dashboard to visualise the data and algorithms to simulate a working energy trading platform based on the data provided from the end nodes.

1.2 Ethical/Commercial/Safety/Environmental

There should be no negative ethical concerns in the pursuit of the project. The purpose of this project is to attempt to advance Ireland’s renewable energy initiative by stabilising the electric grid from unpredictable factors which may cause blackouts or brownouts. Therefore, the ethics in this project is for a positive cause.

This final year project has prompted me to help with a company called Smart M Power. Smart M Power is a Research and Development company that has undertaken multiple projects concerning smart grids. The final year project would have support from the business in the form of materials and equipment, advice and expertise. Any specific involvement in the form such as expertise, code

etc. shall be acknowledged in the document. Technologies, methodologies or code utilised in this project may be used for commercialisation purposes for Smart M Power.

While the project involves electric grids with smart batteries and grid-tie inverters within different floors in a building, this project only seeks to create a simulation of them. Therefore, there would be no direct handling of high-voltage electrical equipment. So, there is no need, for example, to have a qualified professional with the necessary credential to install a grid-tie inverter.

There should be no adverse environmental impact in the project, except for perhaps the electricity used to power the equipment. Non-renewable resources are still the largest energy contributor to the electrical grid, in comparison, Ireland's largest renewable energy contributor is wind power which only accounts for 86%of total renewable electricity and 36% of total electricity demand.

The project safety plan has already been filled in. The potential hazards that are identified in the document are:

- Computer Usage
- Lone Working
- Working Off Campus / Field Work

The category of supervision ticked in the document is Category D. Which means that the risks are low and carry no special supervision requirements.

1.3 Literature Review

The world in recent times, has started to shift towards green energy generation resources such as solar, wind, tidal etc. This can be seen in international politics such as The Paris Agreement which is a legally binding international treaty to tackle climate change. 196 parties such as Ireland hinged into an agreement in 2016 which has encouraged them to pursue carbon neutrality targets. The Irish government has further legally bound itself by signing the "Climate Action and Low Carbon Development (Amendment) Act 2021" where Ireland has to undertake a path towards net-Zero emissions by 2050 and have a 51% reduction in emissions before 2030.

One of the biggest problems faced by renewable energies is that they are produced at certain times of the day and are typically not meeting peak demand hours. For example, solar PV produces energy during the day but will not be able to meet energy demand during the night when consumers return to their homes. Solar energy and wind energy are unpredictable which can cause instability in the grid via blackouts or brownouts.

One way to handle this is the use of a battery storage system. It allows the grid the ability to store the excess energy for the energy to be used at a later time. Though the cost and capacities of batteries have improved over time, large-scale deployment battery storage systems are still too expensive to be feasible. The other method is smart grids.

A smart grid is an electrical grid that provides multiple services that are normally not provided by traditional grids. Some of these services are:

- Advanced metering infrastructure i.e., smart meters.
- Demand response which is a change in power consumption of the electric utility to match the demand for power with the supply.
- Use of renewable energy resources such as charging electric vehicles, batteries etc.
- Export of surplus energy produced by non-electric utilities such as businesses and households or in other words, prosumers.

- The utilisation of wired or wireless communication technology to connect and monitor the various technologies mentioned.

1.4 Work Plan

As mentioned earlier, there are three main stages in this project, creating a LoRa 2.4 GHz Network, a simulated PicoGrid and the creation of an Energy Trading Platform. There is a good chance that the Energy Trading Platform would not have sufficient time left remaining to be worked, so it would it apt to consider that stage an optional objective.

The first stage, the creation of a LoRa 2.4 GHz Network has the following activities that need to be done:

- Research LoRa 2.4 GHz and LoRaWAN
- Change the firmware on the end nodes
- Programme the end node to transmit uplink data
- Install ChirpStack on the gateway and configure the gateway to set up the end nodes to the gate
- Programme gateway to transmit downlink data
- Work on documentation on setting up the gateway and end node
- Perform coverage planning using software such as CloudRF
- Perform testing scenarios (reliability, coverage)

The second stage which is creating a simulation of the Pico-Grid requires:

- Research Smart Meters and Smart Inverters
- Create a Modbus bridge between the LoRa end node and an Arduino device acting as the Smart Metre
- Create the Modbus slave to mimic the Smart Metre's input register
- Make the LoRa end node transmit the Smart Metre data to the gateway
- Figure out how to implement a mimicry of a Smart Inverter and exchange data between it and the LoRa end node. May have to be Modbus or Sunspec Modbus
- Make the LoRa end node transmit the inverter data to the gateway and the gateway sends downlink information back to the device mimicking the inverter

The third stage is creating an energy trading platform which would need:

- Research energy trading platform/peer-to-peer energy trading
- Setup a server/database on the laptop to store data
- Retrieve data received from the ChirpStack network
- Programme trading algorithms
- Develop the GUI software of the energy trading platform
- Come up with a unit test for the software and the trading algorithm

Chapter 2: Design Development

2.1 Overall Network Design

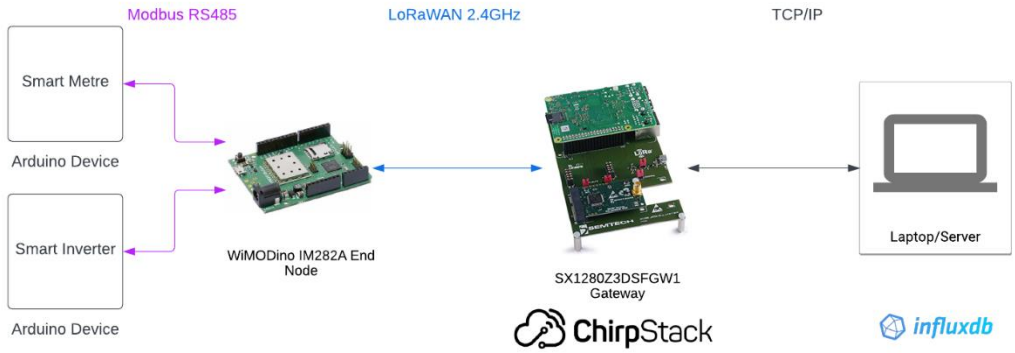


Figure 1: Overall Network Design

This project will be using three main communication standards, which are Modbus RS485, LoRaWAN 2.4 GHz and TCP/IP. There are two Arduino Nanos that would be acting as a smart metre or a smart inverter and they would be connected with their own WiMODino iM282A via Modbus RS485 to exchange data. The WiMODinos will communicate with the SX1280Z3DSFGW1 ChirpStack gateway using LoRa 2.4 GHz. Then finally, the data will be sent to the laptop's InfluxDB database through TCP/IP or more specifically HTTP.

The data will be able to be observed through InfluxDB's dashboards using the localhost. There is also a Python program that is connected to the InfluxDB to get data and it is also connected to the ChirpStack network to send downlink communication back to the WiMODinos. The Python program should have been able to make decisions based on the retrieved data and then downlink data back to the end nodes. For example, if the smart inverter has detected that the connected battery has almost 10%, the Python code then sends a downlink a battery shutdown message to the inverter to avoid damage to the battery.

2.2 Hardware Equipment

2.2.1 SX1280Z3DSFGW1

The SX1280Z3DSFGW1 was originally intended for use with The Things Network (TTN) as stated in the "User Guide to the LoRa 2.4GHz 3 Channels Single SF Reference Design" which is its quick start guide. For the purposes of the project, I would be using the ChirpStack network.

Figure 2: SX1280Z3DSFGW1

2.2.2 WiMODino iM282A

The WiMODino is an Arduino board that is made for fast prototyping and evaluation of the iM282A LoRa radio module. The iM282A is a cheap radio module that uses the unlicensed 2.4 GHz frequency band with the SX1280 transceiver from the Semtech Corporation.

The WiMODino has accidentally been ordered with the LR Base Plus firmware which is a proprietary software that allows LoRa to implement features not found in LoRaWAN such as a LoRa gateway and a network server not being necessary and the use of a peer-to-peer or star network topology. However, this technology is outside of the project's scope. Therefore, the software needed to be replaced with another software called GlobalLink24 which is compatible with the LoRaWAN.

Figure 3: WiMODino IM282A

2.2.3 Arduino Nano

The Arduino Nano is perhaps the second most popular microcomputer from Arduino. It is a relatively small and breadboard-friendly board that is based on the ATmega328P microcontroller. It has a lot of the same features as the Arduino UNO but it is a cheaper alternative.

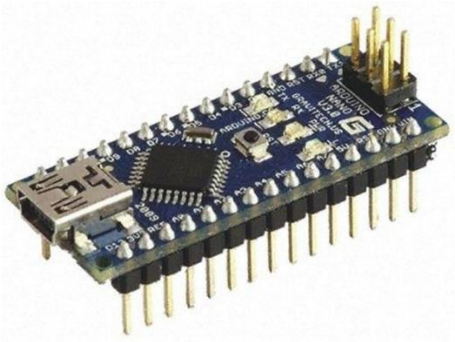


Figure 4: Arduino Nano

2.2.4 LTC1485

The LTC 1485 is a differential bus transceiver that is designed to facilitate communication between two different signals by converting the electrical signals into using either the RS422 or the RS485 point-to-point communication standard. In this project, I am using the RS485 to allow communication between the Arduino Nano and the WiMODino iM282A.

Figure 5: LTC1485

2.2.5 Physical Setup

The image below is a photo of the physical setup between WiMODino which is connected to the Arduino Nano acting as a smart metre. The other Arduino Nano will be emulating as the smart inverter.

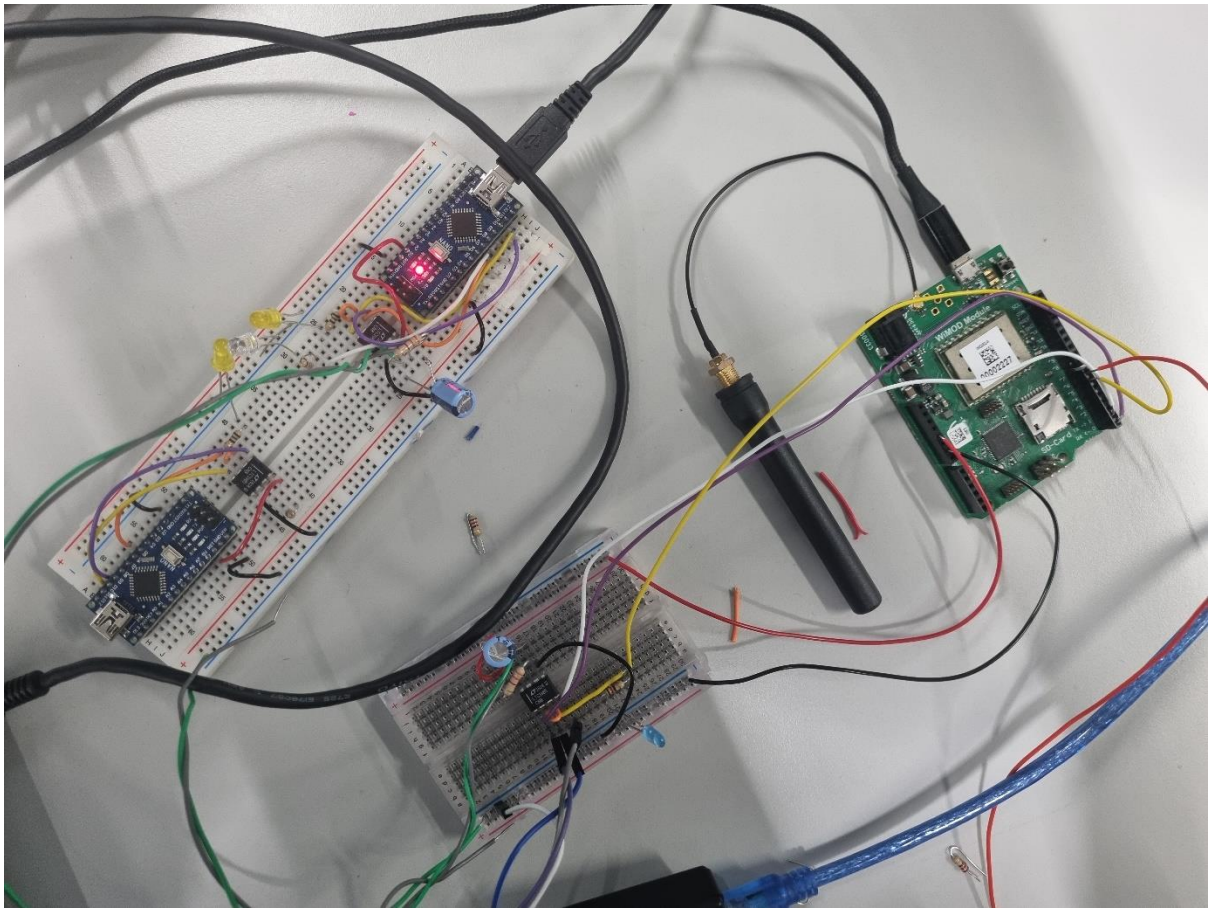


Figure 6: WiMODino & Arduino Nano

2.3 Software

2.3.1 LoRa

The LoRa wireless technology is a proprietary technology developed by Cycleo and acquired by Semtech in 2012. Low Power Wide Area Network (LPWAN) technology allows for long-range communication with low power consumption. LoRa achieves its long-range connectivity by using a modulation technique called Chirp Spread Spectrum (CSS), which spreads the signal over a wide bandwidth and makes it more robust against interference and noise.

LoRa is designed to transmit small payloads like sensor data, making it suitable for Internet of Things (IoT) applications. It is a half-duplex communication technology, which means that it can send and receive data, but not at the same time. LoRa uses a star network topology or a star-to-star network topology, which enables it to communicate with multiple end nodes through a single gateway.

The minimum received signal strength indicator (RSSI) of LoRa is -120 dBm, which is the minimum signal strength required for successful communication. The LoRa technology operates at the physical layer of the Open System Interconnection (OSI) model layer.

It is important to note that LoRa is often confused with LoRaWAN, which is a protocol that operates at the data link and networking layers of the OSI model. LoRaWAN is an open standard that allows for interoperability between different LoRa devices and networks, while LoRa is a proprietary technology with a closed source.

Finally, it is worth noting that for this project, battery optimization may not be a significant concern as both the gateway and end nodes will be powered via mains.

2.3.2 LoRa Modulation

CSS is known for low power consumption and robustness against channel degradation challenges such as interference and multi-path fading. It is a spread spectrum technique that uses wideband linear frequency-modulated chirp pulses to encode data. Spread spectrum techniques are methods by which a signal is deliberately spread in the frequency domain. For example, a signal is transmitted in a short burst, "hopping" between frequencies in a pseudo-random sequence.

LoRa nodes have a radio module with a modulator that encodes information onto a carrier signal. This modulated signal is transmitted and received by a gateway or end node. The LoRa nodes also have a demodulator which decodes the modulated signal and extracts the information.

Figure 7: End node to gateway modulation signals, source at [8]

A chirp, often called a sweep signal, is a tone in which the frequency increases (up-chirp) or decreases (down-chirp) with time.

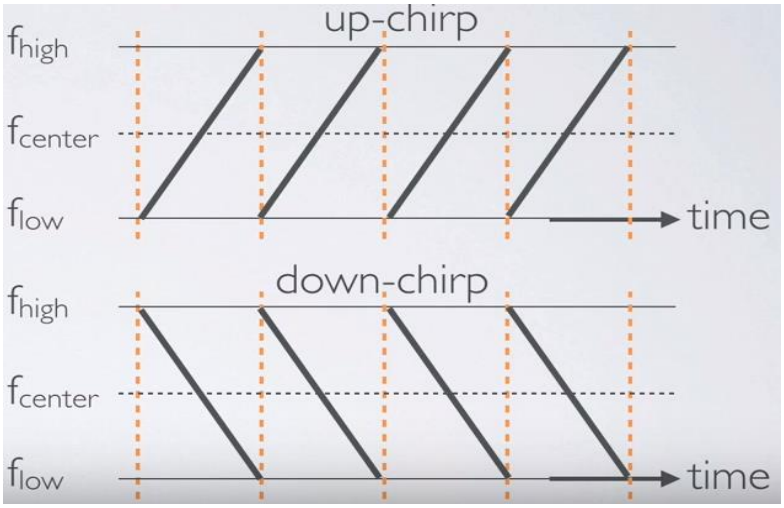


Figure 8: Sweep signal, source at [9]

The chirps are cyclically shifted, and it is the frequency jumps that determine how the data is encoded onto the chirps, aka LoRa modulation.

The spreading factor (SF) defines two values:

- The number of raw bits that can be encoded by that symbol
- Each symbol can hold 2^{SF}

Spreading Factor impact:

- The symbol duration doubles compared to the previous SF
- It reduces the bit rate by approximately half compared to the previous SF

- The Time on Air (ToA) increases which means the distance increases. ToA is the message transmission time

If an end device is further away from a gateway the signal gets weaker and therefore needs a higher spreading factor.

A symbol represents one or more bits of data, for example, Symbol = 1011111 (decimal = 95). In this example, the number of raw bits that can be encoded by the symbol is or has a Spreading Factor (SF) of 7. The symbol has 2^{SF} values. If SF =7, the values range from 0-127 Therefore the chirp is divided into 2^{SF} steps. The Spreading Factor values are 7 – 12. The higher the value, the longer the range but lower data rates and longer ToA.

Note: Difference between chirp & chip:

- A symbol holds 2^{SF} chips (To make things easier, symbols and chips can be used interchangeably).
- Chirps are simply a ramp from low to high signals (down-chirp to up-chirp).

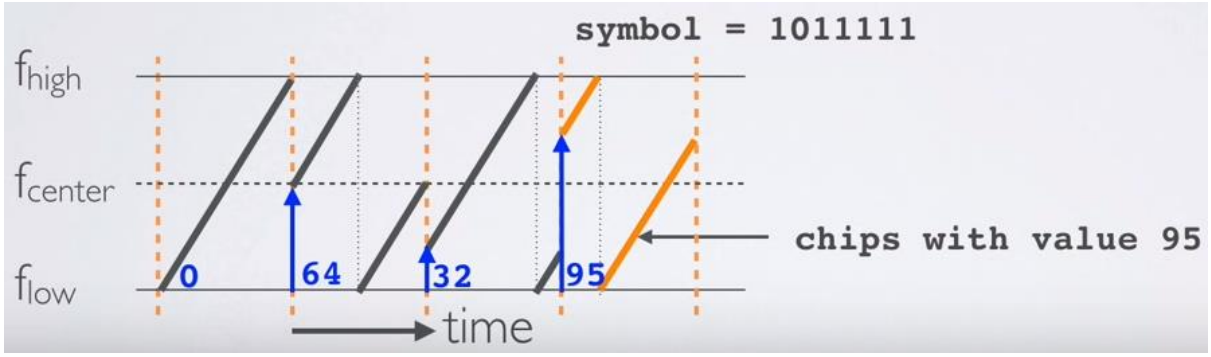


Figure 9: Sweep signal divided into 2^{SF} chips, source at [9]

For each spreading factor, there is a SNR limit, if this limit is reached the receiver will never be able to modulate the signal.

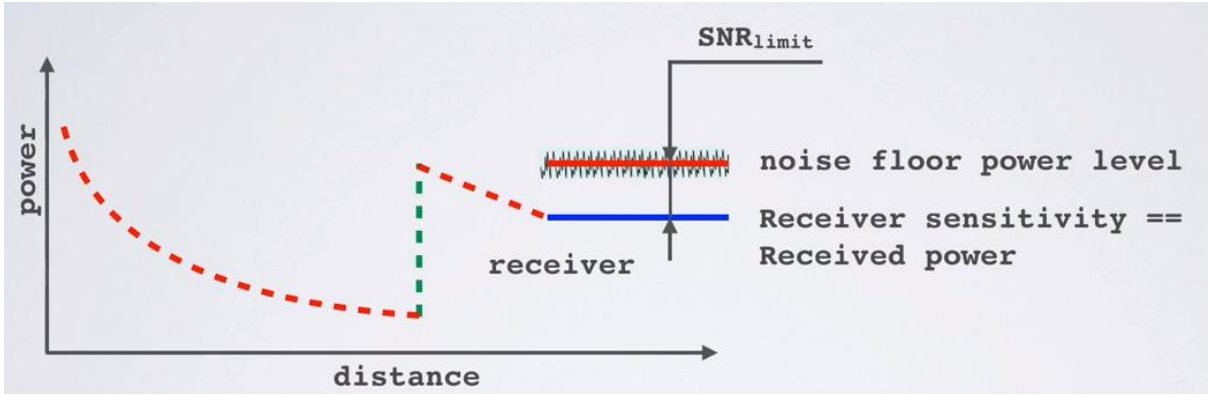


Figure 10: SNR limit, source at [9]

LoRa has a Forward Error Correction (FEC) function where error correction bits are added to the transmitted data. These redundant bits help to restore the data when the data gets corrupted by interference. If more error correction bits are added, the easier the data can be corrected. However, adding more can decrease battery life.

Coding Rate (CR) refers to the proportion of the transmitted bits that carries information, the rest being FEC bits. The LoRa coding rate values are CR = 4/5, 4/6, 4/7 or 4/8.

The LoRa packet comprises three elements: Preamble, header (optional) and payload.

Figure 11: LoRa packet structure, source at [10]

The preamble is used to synchronise the receiver with the incoming data flow. By default, configured with a 12-symbol long sequence. This is shown as the unmodulated data signal the and preamble ends with the 2 down-chirps.

Explicit header mode includes a short header that contains information about the number of bytes, coding rate and whether a 16-bits CRC is used in the packet. This is the default mode.

Implicit header mode is where the payload, coding rate and CRC presence are fixed or known in advance. This will remove the header from the packet. The known parameters must be configured on both sides of the radio link. This mode will reduce transmission time.

The payload is a variable length field that contains the actual data coded at the error rate (Coding Rate) either as specified in the header in explicit mode or the register settings in implicit mode.

2.3.3 LoRaWAN

While LoRa is on the Physical Layer of the OSI model, LoRaWAN is on the Session, Network and Data Link Layer. LoRaWAN is implemented on top of the LoRa modulation bands. LoRaWAN is an open standard that is defined and maintained by the LoRa Alliance, a non-profit organization that promotes the adoption of the LoRaWAN technology.

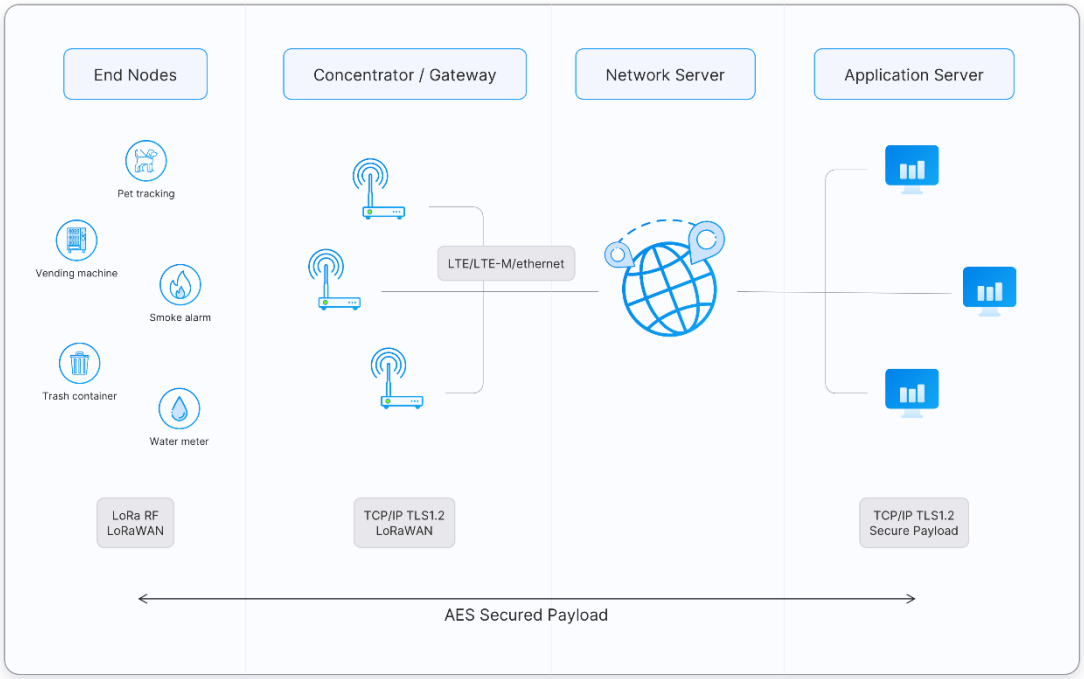


Figure 12: LoRaWAN Architecture, source at [11]

The LoRaWAN protocol provides a range of features that enable efficient and reliable communication between LoRa devices and LoRaWAN networks. These features include:

- Configuring radio parameters: LoRaWAN allows network operators to configure the radio parameters of LoRa devices remotely. This enables operators to optimize the performance of the network and ensure that devices are using the most efficient settings for their specific applications.
- Device activation: LoRaWAN supports two types of device activation, namely over-the-air activation (OTAA) and activation by personalization (ABP). OTAA is a secure and automated method of device activation that involves exchanging keys between the device and the network. ABP, on the other hand, is a manual method of activation that involves pre-configuring the keys on the device.
- Message integrity checking: LoRaWAN includes mechanisms for ensuring the integrity of messages transmitted over the network. This is achieved through the use of message authentication codes (MACs) that are generated by the sender and verified by the receiver.
- Session management: LoRaWAN manages the session between the device and the network, including authentication, encryption, and key management.
- Application payload encryption: LoRaWAN provides encryption of application payloads, ensuring that sensitive data is protected from unauthorized access.

2.3.4 LoRaWAN Summary of Classes and Gateway

Class A Devices:

- Send at any time
- Receive directly after uplink
- Device always initiates communication
- Most battery efficient

Class B Devices:

- Receive at specific time intervals
- Consumes more energy than Class A
- Beaconing

Class C Devices

- Receive at any time
- Consumes most energy
- Continuous listening

2.3.4.1 Class A End Nodes

Class A suitable for:

- Devices that send data on time-based interval
- Send event-driven data (e.g. temperature goes above 20°C)

Each uplink transmission is followed by 2 short downlinks receive windows to allow packets from the server to be received. Downlink messages from the server will have to wait until the next scheduled uplink.

If a message arrives, the end device doesn't know for whom the packet is meant. The packet gets demodulated and the device address and Message Integrity Code (MIC) need to be checked. If the

packet is meant for the device that receives it only opens the first receive window and not the second. If this is not for the device, it reads the first and second windows.

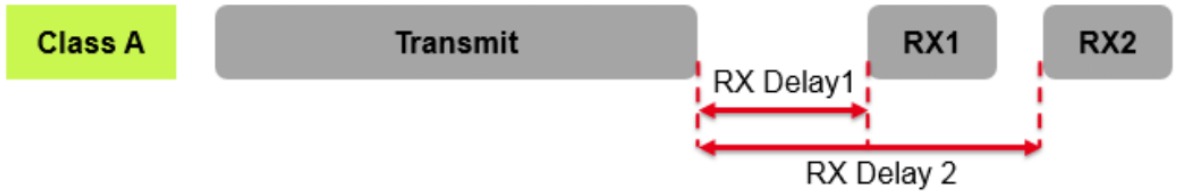


Figure 13 Class A Transmission, source at [12]

2.3.4.2 Class B End Nodes

This allows for more receive slots than for Class A. In addition to their class A receive windows, these devices have extra Rx at scheduled times. To do this, it receives a time-synchronized beacon from the gateway. This allows the server to know when the node is listening. The device must first receive a network beacon and align its internal timing with the network server. The end node can be used by the network to initiate downlink communication. A network-initiated downlink message is called a ping. The gateway selected to initiate this downlink is selected by the network server based on signal strengths and the quality of uplink messages. The device must periodically send a beacon message to cancel any drift that may occur with its internal clock.

Figure 14 Class B Transmission, source at [12]

2.3.4.3 Class C End Nodes

These devices have continuous receive windows that are only closed when the device is transmitting. They consume the most power. They implement the same 2 receive windows as Class A, but they don't close Rx2 until they need to send again. This way they can receive the downlink in the Rx2 window at any time. There is no specific message sequence for the node to tell the server whether it is Class A or C. It is up to the application server to be aware that it manages Class C devices based on the characteristics during the join process.

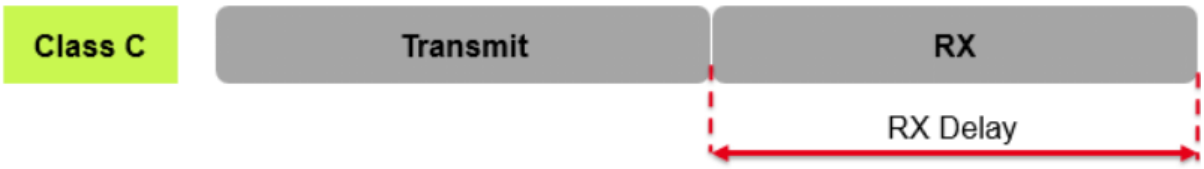


Figure 15 Class C Transmission, source at [12]

2.3.5 Network Server Key Features

The Network Server (NS) is responsible for managing the connectivity of devices and gateways. Some key features of a LoRaWAN Network Server include:

- Connection with gateways: The NS communicates with LoRaWAN gateways to receive data from devices and send data to them.

- Commissioning and supervision interface: The NS provides an interface for commissioning new devices and supervising the network.
- Deduplication of packets
- Device registry: The NS maintains a registry of devices connected to the network, including their security sessions, operation modes, timing windows, frequencies, and data rate for downlink.
- Device identification: The NS uses the network session integrity key to identify devices
- Control data rate: The NS uses Adaptive Data Rate (ADR) to control the data rate of devices to optimise network performance and battery life.
- Set radio parameters and timing windows: The NS configures radio parameters and timing windows for devices, optimising network performance.
- Prevent replay attacks
- Downlink message queue: The NS maintains a queue of downlink messages to be sent to end node devices.
- Select gateway for downlink: The NS selects the best gateway for sending downlink messages to devices based on Link Quality Indication, which includes factors such as Received Signal Strength Indicator and Signal-to-Noise Ratio.
- Prevent conflicting messages: The NS prevents conflicting messages from being transmitted by devices.

2.3.6 Application Server Key Features

The Application Server (AS) is in charge of processing the application data sent by devices. Some key features of a LoRaWAN Application Server include:

- Connected to the Network Server: The AS is connected to the NS which enables it to receive and process application data sent by devices.
- Encryption and decryption of payload: The AS is responsible for encrypting and decrypting the payload sent by devices.
- Recommended to run in the user's domain: The AS is typically run in the user's domain, allowing them to have full control over their application data.
- Web interface: The AS may provide a web interface that enables users to access and manage their application data.
- Integrate with IoT platforms: The AS can integrate with various IoT platforms which creates the capability for users to leverage the data sent by their LoRaWAN devices in various applications and services.

2.3.7 Join Server Key Features

The next component in a LoRaWAN network is the Join Server (JS) which is for securely activating devices and setting up their communication sessions with the Network Server (NS) and Application Server (AS). Some key features of a LoRaWAN Join Server include:

- Activation of devices: The JS is responsible for securely activating new LoRaWAN devices and setting up their communication sessions with the NS and AS.
- Session Security Keys: The JS generates and provides the necessary Session Security Keys to the device for secure communication with the NS and AS.
- Specifies which Network Server the User wants to work in: The JS enables the user to specify which NS they want their device to work with.

2.3.8 Adaptive Data Rate (ADR)

Adaptive Data Rate (ADR) enables the Network Server to dynamically control the data rate of devices. Initially, ADR instructs devices to use the highest data rate possible for maximum capacity and the most efficient power consumption. This is achieved by sending ADR commands to devices to keep or increase their data rate.

If the network does not respond to several messages, the device will lower its data rate to increase the probability that packets are received by the network. Additionally, ADR reduces the transmission power to have less channel utilization of gateways.

It is important to note that ADR only works for devices that are not moving, so the network learns over time which data rates work best when gateways and devices are in fixed locations. As a result, ADR provides a mechanism for optimal and efficient use of the available network resources while ensuring reliable communication between devices and the network.

2.3.9 Gateway message processing

Gateway message processing allows gateways to receive and forward messages between end devices and the network server. One of the key features of gateways is their ability to receive messages on different frequency channels and with different data rates at the same time. This means that gateways can simultaneously receive messages from multiple end devices, allowing for efficient use of available network resources.

In addition, gateways provide message acknowledgement for important packets. If a message is not received, the end device will retry sending the message until it receives confirmation of successful transmission. This ensures reliable communication between the end devices and the network server.

2.3.10 LoRaWAN Security

LoRaWAN provides security features to ensure the confidentiality, integrity, and authenticity of the transmitted data between the end devices, gateways, and network servers. These security features are implemented at the network and application layers.

At the network layer, LoRaWAN ensures the identification of the device and integrity check of the transmitted data. Each end device has a unique identifier that is used to identify and authenticate the device. The transmitted data is checked for its integrity to ensure that it has not been tampered with during transmission. This is achieved using a 32-bit Message Integrity Check (MIC) that is calculated by the end device and verified by the network server.

LoRaWAN uses the Network Session Integrity Key to ensure the security of the network layer. This key is unique to each end device and is used to encrypt and decrypt the network layer messages. The 128-bit Advanced Encryption Standard (AES) algorithm is used to encrypt the MAC commands, which are sent over the air interface. These commands can include changing the data rate, enabling or disabling channels, and changing the downlink frequencies, among others.

At the application layer, LoRaWAN ensures the encryption and decryption of the application payload using the Application Session Key. This key is also unique to each end device and is used to encrypt and decrypt the application payload. The 128-bit AES algorithm is used to encrypt the application payload, providing end-to-end security.

2.3.11 Device Activation

LoRaWAN supports two types of device activation methods - Activation by Personalization (ABP) and Over-the-Air Activation (OTAA).

Activation by Personalization (ABP) is a method where the device is pre-configured with its unique device address and security session keys. The device is programmed with these parameters before deployment, and the network server is already aware of the device's identity. This approach is simpler and faster than OTAA since there is no exchange of messages between the device and the network server to establish security keys. However, ABP has some security limitations as there is no exchange of keys between the device and the network server.

Over-the-Air Activation (OTAA) is a more secure activation method, where the device and network exchange a series of messages to establish a secure session. When a device activates by joining the network, it sends a Join Request message to the network server. The network server then responds with a Join Accept message containing a unique device address, network session key, and application session key. These keys are used for subsequent communication between the device and the network server. This method is more secure as the keys are exchanged securely but it takes more time than ABP. OTAA is recommended for most applications where security is a concern.

2.3.12 ChirpStack

To create a private LoRaWAN network with no third-party cloud networks, ChirpStack was used. This is an open-source LoRaWAN Network Server stack provides that the whole LoRaWAN stack including the network and application server. It allows for easy device activation and management, data handling, and integration with various applications and IoT platforms.

ChirpStack provides a web-based interface for managing the gateways. It allows for configuring the gateway parameters, monitoring the gateway status, and troubleshooting any issues. It also provides various APIs and protocols for integrating with external applications and IoT platforms. It allows for easy data exchange and seamless integration with different systems.

The graph below showcases the ChirpStack architecture of the components:

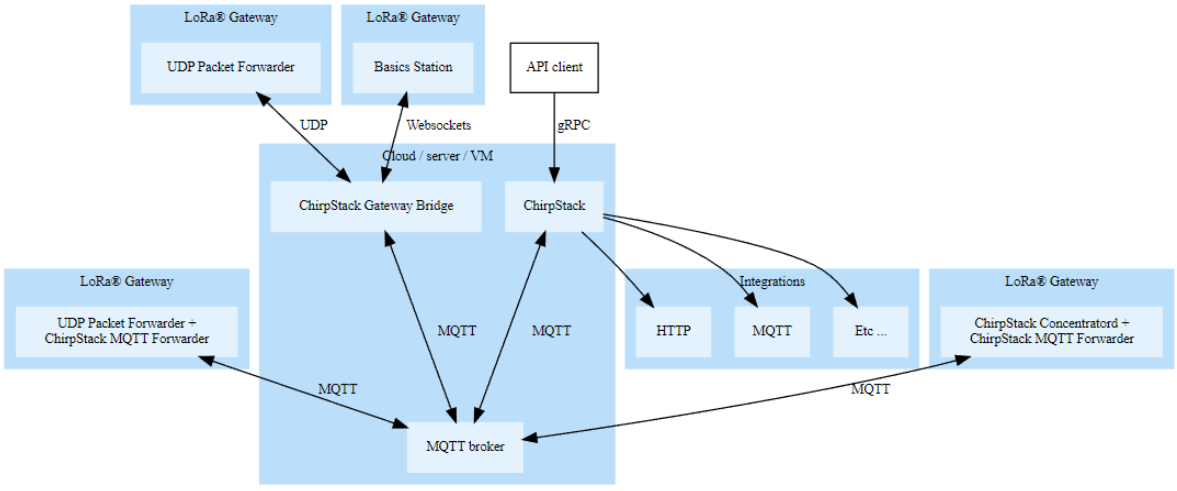


Figure 16: ChirpStack Architecture V4, source at [13]

- The ChirpStack Concentratord is a concentrator daemon that is placed on top of the hardware abstraction layers. It uses ZeroMQ which is an open-source universal messaging library that provides sockets that can send atomic messages in various transports such as TCP, multicast, in-process and inter-process. It could also run without a message broker. The ZeroMQ-based API allows packet wording applications like the UDP forwarder and MQTT forwarder to be decoupled from the gateway hardware.
- The ChirpStack MQTT Forwarder is a component that converts LoRa Packet Forwarder protocols into JSON and Protobuf.

- The ChirpStack Gateway Bridge is a service whose purpose is to convert LoRa packet forwarder protocols into a common data format

2.3.13 Modbus

Modbus was created in the 1970s by a company called Modicon which is now called Schneider Electric as a communication protocol standard for programmable logic controllers [8]. This protocol uses a master and slave technique which allows a master to poll at least one device for data that are operating as a slave. The master can read and even write data on the slave data storage. There are two main Modbus protocols which are Modbus RTU and Modbus TCP. In this project, Modbus RTU is used.

Modbus RTU is perhaps the most commonly used as it is a proportionally simple protocol that transmits data using UART. The data are being transmitted in 8-bit bytes on baud rates that can be around 1200 bits per second (bps) or 38400 bps. In a Modbus RTU network, each slave must be assigned a device address or a unit number for the master to identify on which it should send the message. The slave must send a response on a timely matter or else the master would have to declare it to have a no response error. Modbus RTU which is using the RS-485 network could only have 32 devices.

The Modbus data are defined to be 16-bit registers. However, if a 32-bit floating point is required then those values need to be treated as a pair of registers. There are 4 types of registers;

- Coil/Discrete output: This is a 1-bit register that can be read or written. Typically used as true or false statements.
- Discrete input: This is also a 1-bit register but can only be read.
- Input register: It is a 16-bit register that can only be used to read values.
- Holding register: Just like the input register, it is 16-bit but it can read and write data.

2.3.14 InfluxDB

InfluxDB is an open-source database management system that was developed by InfluxData, Inc in 2013. It is written in Google's programming language called Go or known as Golang. InfluxDB is a type of database called a time series database which is made efficiently to store time-based values. In other words, its database is used to store and analyse sensor data with timestamps over a period of time. To make sure that time is synchronised between systems, InfluxDB is included to have the Network Time Protocol.

InfluxDB's columns have two types, tag and field. The tag column can be used as an identifier or metadata while fields contain the values to be analysed. Even its terminology for retrieved data is called a measurement.

In comparison to standard relational databases such as MySQL, time series databases are faster in storing and processing time-based measurement data. As typical data database systems have more complex indexes which result in slower speed. InfluxDB's more focused or rather simple index allows it high write speeds.

Chapter 3: Implementation

3.1 Flowchart

Figure 17 is a flowchart of the overall system. At the start, the Arduino Nano which can either be acting as the Smart metre or inverter as the Modbus server, will be updating Modbus registers in which the values reside. It will be doing this action every 100ms by updating. The WiMODino which

is the Modbus client will be reading the voltage, current, kWhr and phase and then transmitting via LoRa 2.4 GHz every 30 seconds to the SX1280Z3DSFGW1 gateway. The ChirpStack network which resides on the gateway uses the InfluxDB integration to create a HTTP connection to connect with a laptop. A Python code on the laptop is polling the InfluxDB database every 30 seconds for the latest data and is sending downlink messages to be queued to the ChirpStack network.

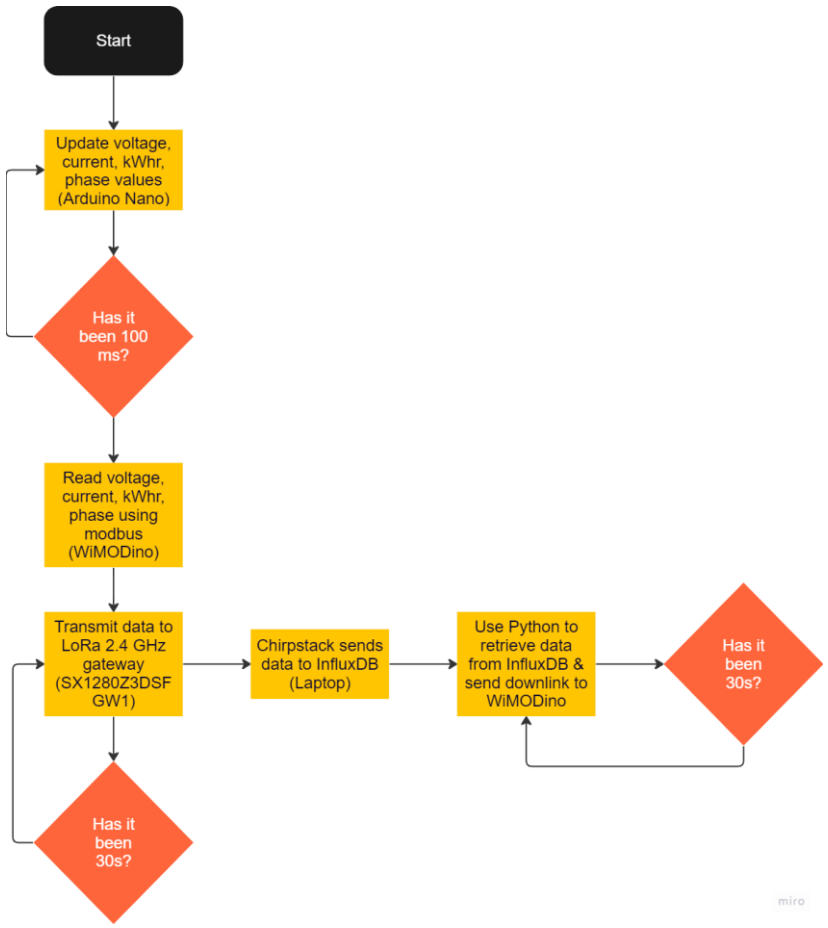


Figure 17: Flowchart of the overall system

3.2 Circuit Diagram

Figure 14 shows the circuit diagram connection between the Arduino Nano, the WiMODino iM282A, the LTC1485 differential bus transceiver, capacitors, LED lights and resistors. The two Arduino Nanos are connected as an RS485 serial bus with the help of the ltc1485 to create an RS485 system. They both have their LTC1485 to signify that they are the es. Meanwhile, the WiMODino has its LTC1485 to act as the master. The yellow LEDs next to each of the Arduino Nanos shows the user that a message has been read. The blue LED connected next to the WiMODino shows that a message has been sent to a slave. The RS485 connection is half-duplex and there should be done on the software side to avoid potential collisions. It should be noted that the physical circuity was set up by Frank Duignan.

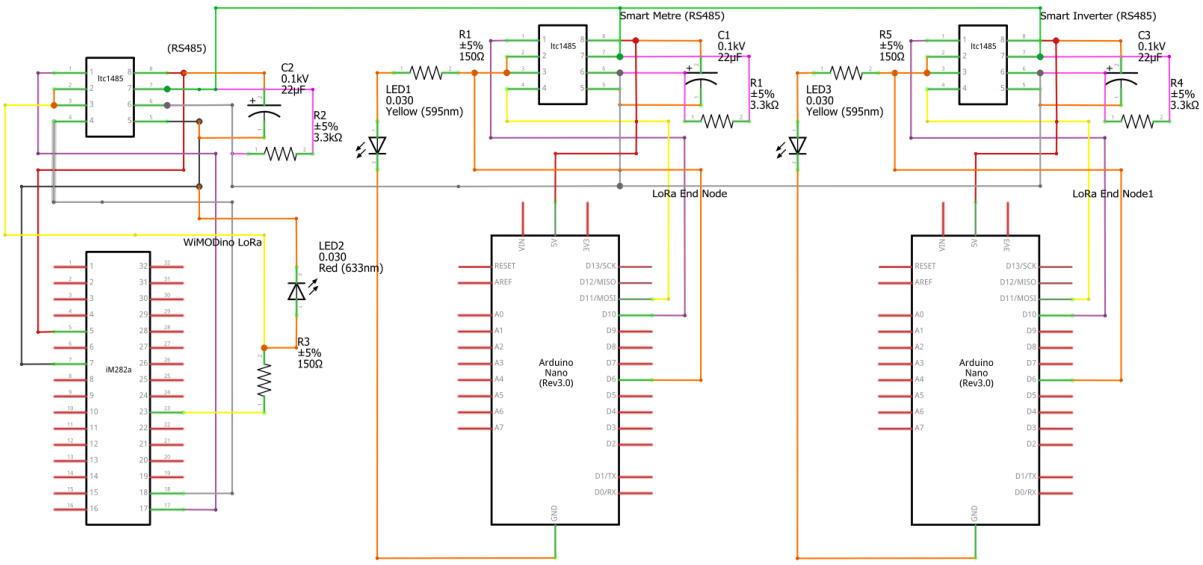


Figure 18: Circuit diagram of WiMODino and Arduino nano (Smart metre)

LTC1485 pin functions and connections:

Pin	Function	Arduino Nano	WiMODino
R0 (Pin 1)	Receiver output	Connected to Arduino Nano's D10	Connected to WiMODino's D1
RE (Pin 2)	Receiver output enable	Connected with DE with Arduino Nano's D6 and LED	Connected with DE with WiMODino's D6 and LED
DE (Pin 3)	Driver output enable	Connected with RE with Arduino Nano's D6 and LED	Connected with RE with WiMODino's D6 and LED
DI (Pin 4)	Driver input	Connected to Arduino Nano's D11	Connected to WiMODino's D0
GND (Pin 5)	Ground Connection	GND pin	GND pin
A (Pin 6)	Driver Output/Receiver Input	This should be connected to another LTC1485	This should be connected to another LTC1485
B (Pin 7)	Driver Output/Receiver Input	This should be connected to another LTC1485	This should be connected to another LTC1485
Vcc (Pin 8)	Positive Supply, 5V	5V pin	5V pin

3.3 CloudRF: Radio Planning Tool

CloudRF has a web application that can be used to help with RF planning. It shows the cover range of radio technologies like LoRa with Google Maps. This was done by inputting the parameters of the gateway like its transmission height, frequency, transmission power, terrain type etc. The terrain data, antenna patterns and clutter layers are all already included. However, it doesn't take account into the spreading factor.

It also has a feature called the Path Profile Analysis (PPA) that shows a point-to-point link between the gateway and the end node on a 2D profile graph highlighting obstructions. An important detail

to when looking at the PPA is the Path Loss. If the path loss exceeds LoRa's receiver sensitivity then communication between the devices is impossible.

3.3.1 Limerick Georgian Building

The building chosen for testing is a Georgian building in Limerick that was chosen as the gateway location for use by Smart M Power. The parameters listed below are what was used for the CloudRF tool.

Site/Tx

- Height AGL: 5 Metres

Signal

- Frequency: 2402 MHz
- RF Power: 28 dBm
- Bandwidth: 0.2 MHz

Feeder

- Coaxial type: TMS LMR-400
- Coaxial length 1m
- Connectors 1

Antenna

- Pattern: Ubiquity AM-2G16-90-HPOL.MSI
- Polarisation: Vertical
- Antenna Gain: 16 dBi
- Azimuth: 0
- Down-tilt: 0

Mobile/Rx

- Receiver Height AGL: 5 Metre
- Antenna Gain: 16 dBi
- Sensitivity: -97 dBm
- Noise floor: -120 dBm

Model

- Propagation model: ITM/Longley-Rice (< 20GHz)
- Reliability: 90% (Higher percentages give more conservative results)
- Context: Average/Mixed
- Diffraction: Knife edge

Clutter

- Profile: Minimal.clt
- Landcover: Enabled + Buildings
- Custom clutter: Disabled

Output

- Resolution: 20m / 65ft
- Colour schema: RAINBOW.dBm
- Radius: 3 Km

As expected, from the results shown in Figure 19 it theoretically be able to provide coverage in a couple of blocks at best.

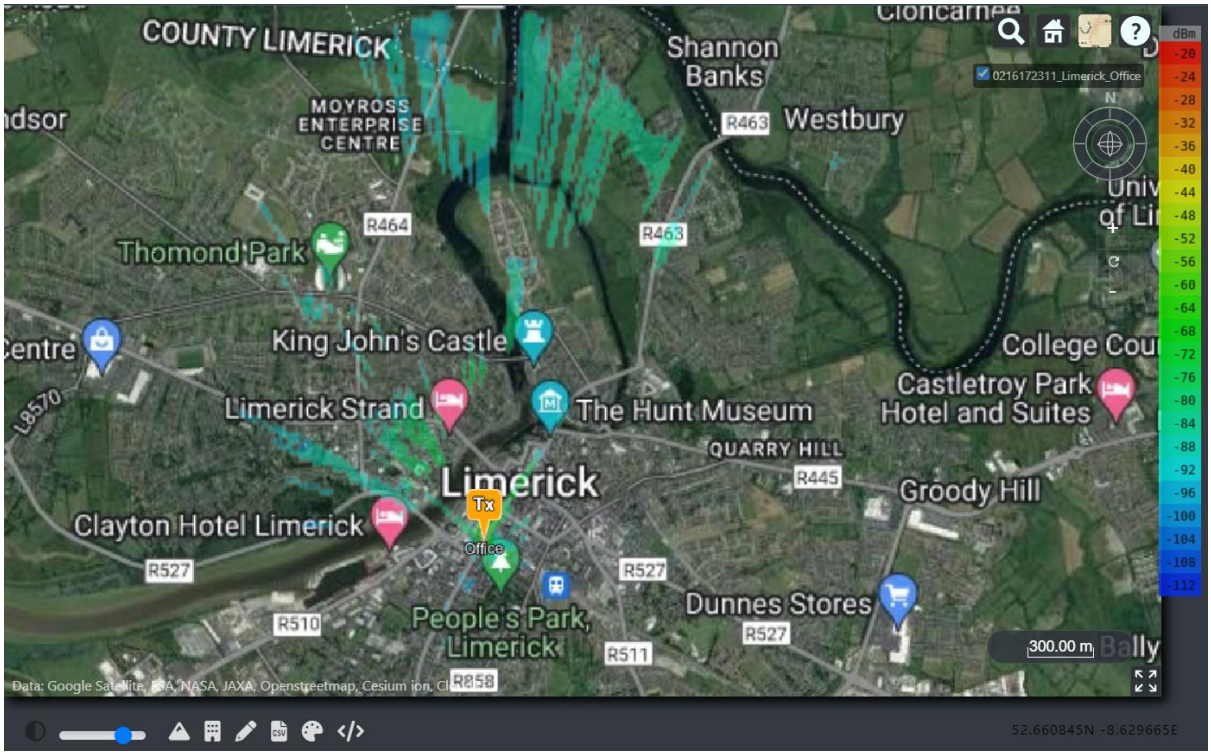


Figure 19: Coverage Map of a Georgian building to be used as a gateway

The end node location is also chosen by Smart M Power as it may potentially set up a LoRaWAN location there. It is merely right across the Georgian building and therefore its RSSI is still fairly strong.

Figure 20: Path Profile Analysis from the Georgian building to Hanratty's Accommodation Centre

3.3.2 TUDublin Central Quad

A place where some actual coverage testing was done was on the fifth floor of TUDublin's Central Quad building. Below are its parameters:

Site/Tx

- Height AGL: 25 Metres (a guess made by Frank Duginan)

Signal

- Frequency: 2402 MHz
- RF Power: 28 dBm
- Bandwidth: 0.2 MHz

Feeder

- Coaxial type: TMS LMR-400
- Coaxial length 1m
- Connectors 1

Antenna

- Pattern: Ubiquity AM-2G16-90-HPOL.MSI
- Polarisation: Vertical
- Antenna Gain: 16 dBi
- Azimuth: 0
- Down-tilt: 0

Mobile/Rx

- Receiver Height AGL: 5 Metre
- Antenna Gain: 16 dBi
- Sensitivity: -97 dBm
- Noise floor: -120 dBm

Model

- Propagation model: ITM/Longley-Rice (< 20GHz)
- Reliability: 90% (Higher percentages give more conservative results)
- Context: Average/Mixed
- Diffraction: Knife edge

Clutter

- Profile: Minimal.clt
- Landcover: Enabled + Buildings
- Custom clutter: Disabled

Output

- Resolution: 20m / 65ft
- Colour schema: RAINBOW.dBm
- Radius: 3 Km

In comparison to the building in Limerick, the Central Quad has a pretty favourable simulated result. This is likely due to the building's higher height which would allow it to go over many obstacles.

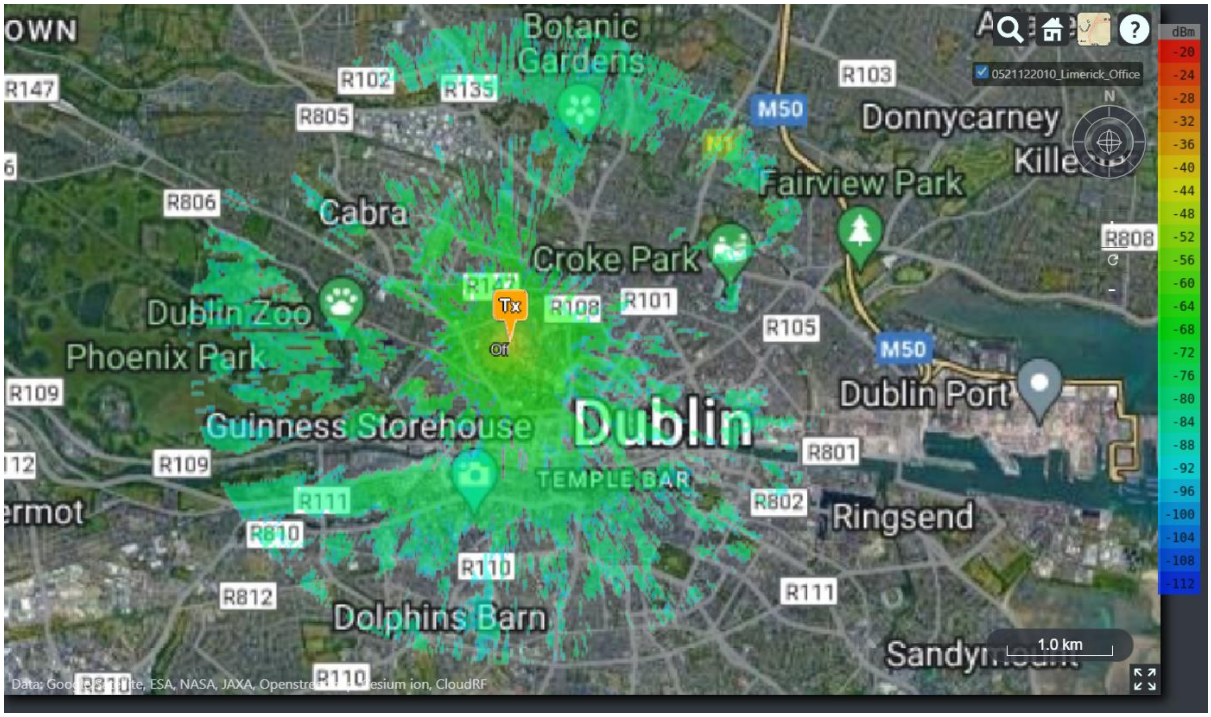


Figure 21: Coverage map of TUDublin's Central Quad

The end node location first chosen to test the coverage range is the nearby Grangegorman Playground. With a clear line of sight, there should be no problem with getting a reliable connection to the gateway.

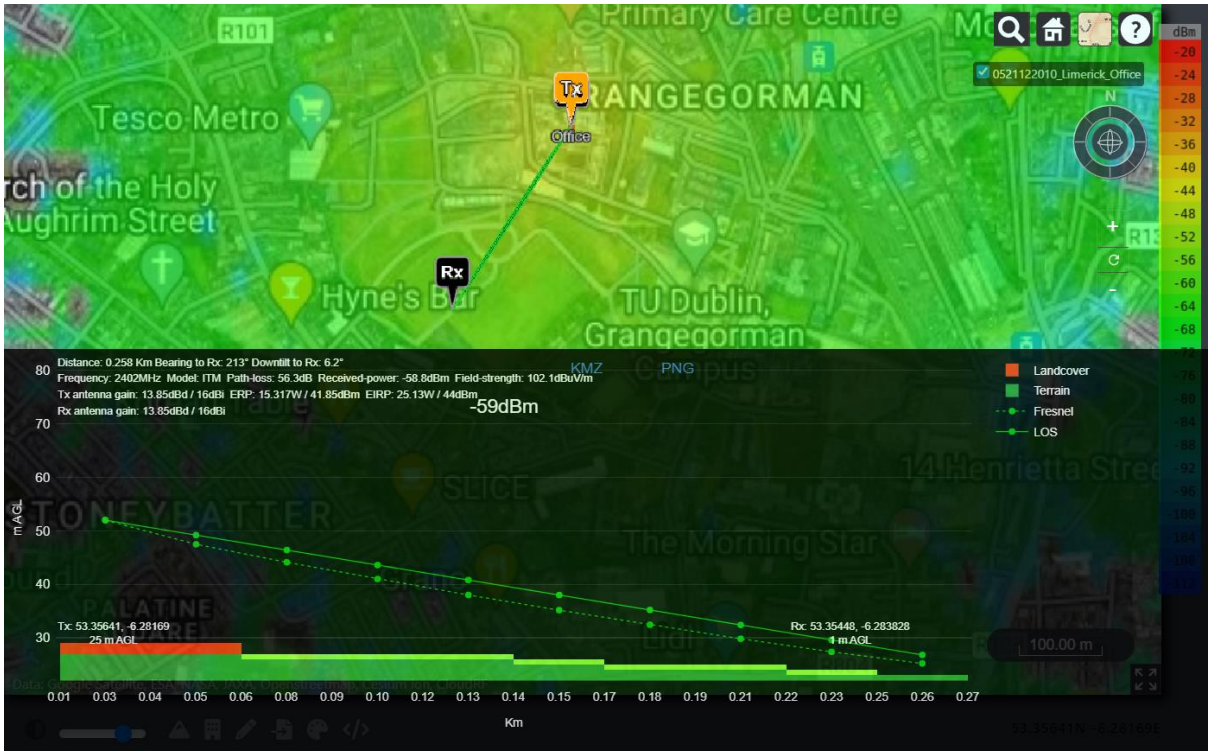


Figure 22: Path Profile Analysis nearby the Grangegorman Playground

4.4 Brief Program Explanation

4.4.1 Arduino Nano codes

The name of the program for the Arduino Nano used to imitate the smart metre is called "ModbusExampleSlave" whilst the one for the smart inverter is the "ModbusExampleSlave2". Both programs are using the SoftwareSerial and SimpleModbusSlaveSoftwareSerial libraries to enable the connection between the ltc1485 and the WiMODino through Modbus. It is in this code that the pins are defined, the data are initialised with data and are placed in Modbus Holding register addresses. In the main loop it checks if there are any errors and updates the values in the Modbus registers every 100ms. "ModbusExampleSlave" has been assigned the slave ID of 1 whilst "ModbusExampleSlave2" has been given slave ID 2. These slave IDs are needed by the master to communicate with them separately.

4.4.2 WiMODino iM282A code

The WiMODino's code is called "WimodModbusClientV2". One of the libraries in use is WiMODGlobalLink24 which enables LoRaWAN communication. It also uses the ArduinoModbus library which depends on the ArduinoRS485 library to establish Modbus RTU communication. Similarly, the pins have to be defined but the pins are defined by default using the ArduinoRS485 library or more specifically the "RS485.h" source file. It initialises all of the needed data variables from both Arduino Nanos.

The WiMODino is configured to use Class C as the WiMODino does not need to conserve power as it will be connected to the laptop for electricity. It is also configured to use the APPKEY and the APPEUI which would provide access to the ChirpStack network. Some important things to note is that the APPEUI is left empty due to the ChirpStack network does not require this but the WiMODGlobalLink24 still requires this to be included.

On the main loop, every data register from both Arduino uses its slave ID to specify them individually. Each Modbus read has been given a 100ms delay for the RS485 time more than enough to read the data. Once all data has been read, they would be combined into four variables that would be transmitted every 30 seconds. All downlink communication is read as hexadecimal values.

4.4.3 ChirpStack to InfluxDB configuration

All received data from the end nodes are in hexadecimal format which will present a problem when trying to send the data to InfluxDB as it would not be in a human-readable format. Within the ChirpStack web interface, there is a way to manipulate all uplink and downlink data using JavaScript. This is located in the "Device Profiles" section, choose the "WiMODino" device profile and then click "Codec". In summary, the code written would turn the hexadecimal payload into a string format. Next, it would divide the payloads into separate variables to enable InfluxDB to appropriately measure the variables and show them in dashboards.

4.4.4 Laptop Python

The laptop runs a Python program called "chirpstackDown.py" which would connect to InfluxDB and retrieves recent data in the last 30 seconds to be printed out on a console. Its functionality is that it would connect to the ChirpStack network using an API token and then send out a downlink message using the WiMODino's unique device EUI.

Chapter 4 Results

4.1 Testing

Coverage testing

Before committing to any coverage testing, some RF Planning must be done beforehand. A suitable for such is CloudRF which is an online RF planning software. Then we can use real coverage testing on a physical location in Limerick or elsewhere. Afterwards, the results from the coverage testing and the expected results from CloudRF can be compared and contrasted for analysis.

Test Scenarios:

- Have the transmitters send data indoors and outdoors.
- Leave the SLU transmitting for 5 minutes.
- Perform one of the tests in severe weather conditions such as heavy fog (Optional).

4.1.1 Coverage Testing

There was coverage testing performed at the Grangegorman campus, specifically the Central Quad on the CQ-507 room. My supervisor and I have made an educated guess that the height at which I placed the gateway is around 25 metres.

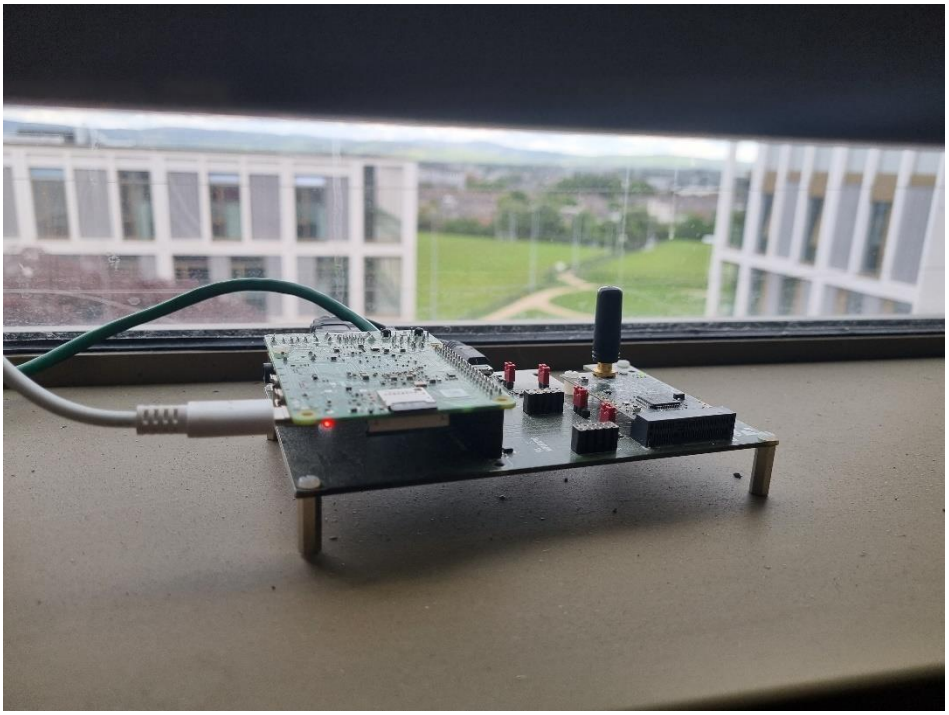


Figure 23: LoRaWAN gateway at Central Quad for coverage testing

The end node location to test is just around 250 metres away from the gateway, nearby the Grangegorman Playground. The end node code has been slightly modified to send 10 data to the gateway every 30 seconds and therefore the test lasts for 5 minutes. Unfortunately, none of the transmissions were detected by the gateway. The likeliest culprit for this may have been the thick double-glazed windows have severely affected the signal's attenuation.

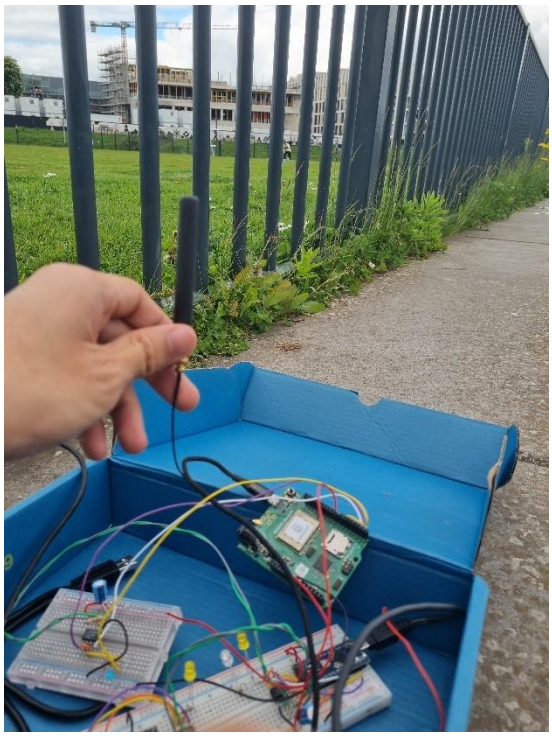


Figure 24: End node nearby the Grangegorman Playground for coverage testing

There was an impromptu testing performed just outside the Central Quad's main entrance which is a mere 50 metres away from the gateway. Puzzlingly the results are just as bad. The next test should be performed clear of obstacles.

4.2 System run

This section shows how the system running looks like and any notable facts that typically occur during its runtime. It may even include an analysis of what may the problem may be.

Running the Arduino Nano programs have no print statements and so the focus will be shifted to the WiMODino. When running the WimodModbusClientV2 program on the Arduino IDE with the console window opened, an example shown in Figure 25 is what is expected to run. Some of the registers read sometimes fail. This could be because the 100ms delays between each read may be insufficient.

```
16:57:52.749 -> Modbus RTU Client Toggle
16:57:52.749 -> ===================================================
16:57:52.749 -> This is FileName: C:\Users\kenne\OneDrive\Documents\Arduino\WimodModbusClientV2\WimodModbusClientV2.ino
16:57:52.749 -> Starting...
16:57:52.749 -> This program will read data through modbus do the OTAA procedure and transmit the read messages every 30 sec.
16:57:52.749 -> downlink messages will be printed in as hex values.
16:57:52.749 -> ===================================================
16:57:52.749 ->
16:57:52.749 -> General Info Radio-Module:
16:57:52.749 -> -------------------------
16:57:52.749 -> DeviceSerial:    0x8B3
16:57:52.749 -> FW-Version:      2.0 BC: 173
16:57:52.749 -> FW-Name:         WiMOD_GlobalLink24;LMIC_V1.5;WW2G4;Modem
16:57:52.749 -> DeviceEUI:       70B3D58FF0040150
16:57:52.749 ->
16:58:17.915 -> Read successful from address 6 v1: 100
16:58:18.009 -> failed to read registers! Invalid CRC
16:58:18.149 -> Read successful from address 8 v3: 100
16:58:18.287 -> failed to read registers! Invalid CRC
16:58:18.428 -> Read successful from address 10 a2: 100
16:58:19.543 -> failed to read registers! Response not from requested slave
16:58:19.684 -> Read successful from address 12 p1: 100
16:58:19.824 -> Read successful from address 13 p2: 100
16:58:19.917 -> failed to read registers! Invalid CRC
16:58:20.056 -> Read successful from address 15 k1: 100
16:58:20.056 -> Read successful from address 16 k1: 100
16:58:20.182 -> failed to read registers! Invalid CRC
16:58:21.302 -> failed to read registers! Response not from requested slave
16:58:21.442 -> Read successful from address 30 vHighGrid: 100
16:58:21.582 -> Read successful from address 31 vLowGrid: 100
16:58:21.721 -> Read successful from address 32 freqHighGrid: 100
16:58:21.858 -> failed to read registers! Invalid CRC
16:58:21.999 -> Read successful from address 34 pfGrid: 100
16:58:22.092 -> failed to read registers! Invalid CRC
16:58:22.234 -> Read successful from address 36 voltageLoad: 100
16:58:22.374 -> Read successful from address 37 currentLoad: 100
16:58:22.469 -> Sending meter message...
16:58:22.469 -> 1_100_0_100_0_100_0
16:58:22.469 -> 2_100_100_0_100_0_0
16:58:22.469 -> 3_100_100_100_0_100
16:58:22.469 -> 4_0_100_100Tx-U-Data Indication: TX OK
16:58:22.469 -> A No-Data Indication has been received --> (timeout for current (re)transmission)
16:58:22.469 -> -> TX Done
```

Figure 25: WiMODino console print

If the end node does pick up a downlink message it gets shown in the console window as well. This is shown in Figure 26.

```
13:28:39.987 -> A No-Data Indication has been received --> (timeout for current (re)transmission)
13:28:39.987 -> Rx-U-Data Indication received.
13:28:39.987 -> Rx-U-Data Message: [3]: 01 02 03
```

Figure 26: WiMODino reading a downlink message

Figure 27 shows a couple of the data received by InfluxDB in a dashboard. Each of the three-phase data is placed within the same dashboard cell, with different colours to differentiate between them.

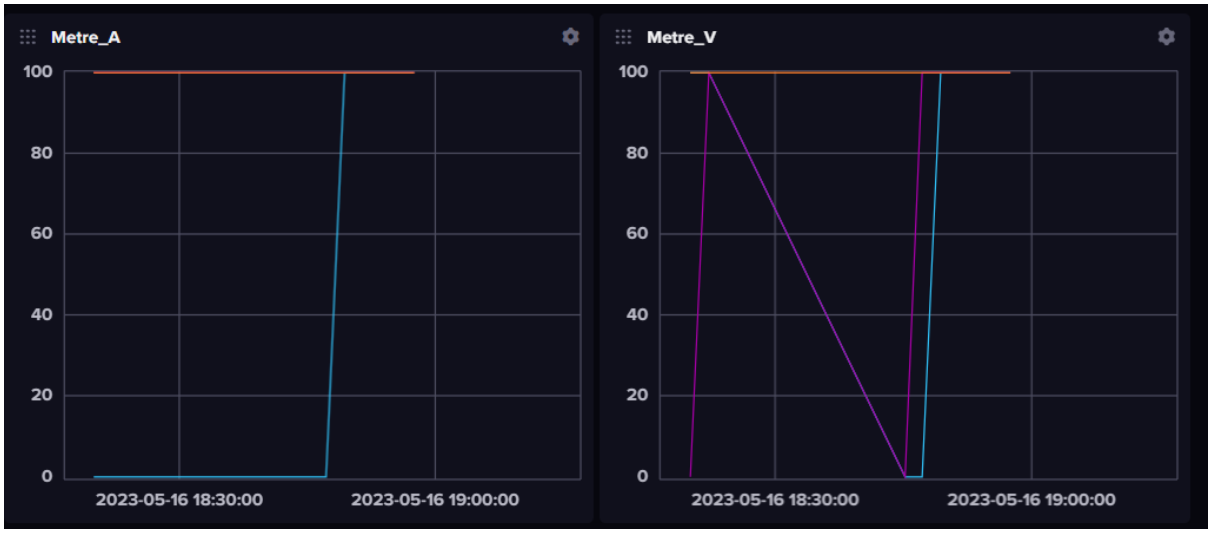


Figure 27: InfluxDB dashboard of metre reads

The IDE to run the Python program is called PyCharm and Figure 28 are a couple of values read by the program when querying InfluxDB every 30 seconds.

```
[('value', 100.0), ('value', 100.0), ('value', 100.0), ('f_cnt', 19), ('rssi', -51), ('snr', 5.0), ('value', 1)]
7ce0891c-8d94-4425-b5d9-06ca8000c311
[('value', 100.0), ('value', 100.0), ('value', 100.0), ('value', 100.0), ('value', 100.0), ('value', 100.0), ('f_cnt', 20), ('rssi', -45), ('snr', 4.0), ('value', 1)]
```

Figure 28: PyCharm IDE console query print to InfluxDB

5.5 Conclusions

5.5.1 Conclusion

Looking at the work plans I had completed stage 1, mostly stage 2 and stage 3 has not even been started. It was expected that stage 3 would not be completed as it has always been seen as a bonus objective just in case, I had managed to breeze through the work plans. Though Stage 2s last objective of making ” ...the LoRa end node transmit the inverter data to the gateway and the gateway to send downlink information back to the device mimicking the inverter” was only partially completed and therefore not to create a simulation of a PicoGrid or any type of smart grids, the main objective given to me by Smart M Power is creating a LoRaWAN 2.4 GHz network has been a success and any further work can be achieved at a later date.

5.5.2 Future Work

Many features and problems have not either been implemented or fixed due to a lack of time and resources. Some of the future work that can be done are:

- **Hardware case**: All of the equipment is fully exposed with no casing to prevent unintentional tampering and the elements such as rain.
- **Different gateway:** The SX1280Z3DSFGW1 is a demonstrator gateway meant for fast prototyping of the LoRaWAN 2.4 GHz could be a problem when used in mass production use. It uses a Raspberry Pi 4 which is quite expensive and is still in short supply worldwide.
- **Real data:** All of the data “read” by the Arduino Nanos are dummy data which may or may not have expected values that should give out in a building. In the future, the Arduino Nano should be physically connected to a smart metre and inverter.
- **Faster end node transmission**: Theoretically the data rate by LoRa 2.4 GHz should be higher than the typical 868 MHz frequency. However, the end node has been failing to transmit data that are less than 30 seconds since the previous transmission. The cause could be that the end node has not been properly configured to ignore Duty Cycle or the gateway has not.
- **Finish stage 2:** As mentioned in the conclusion section, the next big step would be to create the actual PicoGrid network.
- **Finish stage 3:** While not a step that Smart M Power needed me to work towards, this would still be a valuable experience to have. Perhaps something similar can be achieved as a personal project.

Acknowledgement

I would like to thank my supervisor Frank Duignan for providing me with support and guidance. He also helped me by providing me with some of the equipment that was used in this project. Without them, the project would not have progressed as far as it had.

I also would like to thank Smart M Power for supplying some of the other equipment used in the project. Individually I would give sincere thanks to my employer Dudley Steward for not just working on this project but also giving me for the opportunity to continue my education in my fourth year.

The other individual that deserves my thanks is my co-worker Mohammad Niyazi for bestowing me guidance on the Pico-Grid, smart metre and smart inverter.

Bibliography

[1] United Nations Framework Convention on Climate Change. "What is the Paris Agreement?". Internet: https://unfccc.int/process-and-meetings/the-paris-agreement, [Feb. 18, 2023].

[2] Department of the Environment, Climate and Communications. "Ireland's ambitious Climate Act signed into law". Internet: https://www.gov.ie/en/press-release/9336b-irelands-ambitious-climate-act-signed-into-law/, Aug. 30, 2021 [Feb. 20, 2023].

[3] Regen Power. "What are the problems faced by renewable energy?". Internet: https://regenpower.com/articles/what-are-the-problems-faced-by-renewable-energy/#:~:text=Renewable%20energy%20sources%20generate%20most,energy%20and%20wind%20are%20unpredictable, [Feb. 20, 2023].

[4] U.S. Department of Energy. "The U.S. Department of Energy's Microgrid Initiative". Internet: https://www.energy.gov/sites/prod/files/2016/06/f32/The%20US%20Department%20of%20Energy%27s%20Microgrid%20Initiative.pdf, Oct. 2012 [Feb. 24, 2023].

[5] L. Berger, A. Schwager and J. Escudero-Garzás. (2013, March). "Power Line Communications for Smart Grid Applications". *Journal of Electrical and Computer Engineering.* [On-line]. Volume 2013, 16 pages. Available: https://www.hindawi.com/journals/jece/2013/712376/, [Mar. 01, 2023].

[6] G. Wibisono, S Permata, A Awaludin and P. Suhasfan. "Development of advanced metering infrastructure based on LoRa WAN in PLN Bali toward Bali Eco smart grid". Internet: https://ieeexplore.ieee.org/document/8356496, May 10, 2018 [Mar. 02, 2023].

[7] Semtech Corporation. "Academy for LoRaWAN". Internet: https://learn.semtech.com/course/view.php?id=2&section=1, [Apr. 15, 2023].

[8] Mobile Fish. "LoRaWAN". Internet: https://www.mobilefish.com/developer/lorawan/lorawan_quickguide_tutorial.html, [Feb. 18, 2023].

[9] RF Wireless World. "What is difference between Chip and Chirp in LoRaWAN, LoRa". Internet: https://www.rfwireless-world.com/Terminology/What-is-difference-between-Chip-and-Chirp-in-LoRaWAN.html, [Feb. 21, 2023].

[10] Semtech. "SX1272/3/6/7/8: LoRa Modem Designer's Guide AN1200.13". Internet: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001OK4/K1xBJSCPflEbqU03CfABAjL29tRKA9KsdAdTIsWBA8s, Jul. 2013 [Feb 21. 2023].

[11] The Things Network. "LoRaWAN Architecture". Internet: https://www.thethingsnetwork.org/docs/lorawan/architecture/, [Feb 22. 2023].

[12] T. Bouguera, J. Diouris, J. Chaillout, R. Jaouadi and G. Andrieux. "Energy Consumption Model for Sensor Nodes Based on LoRa and LoRaWAN". Internet:

https://www.researchgate.net/publication/326134076_Energy_consumption_model_for_sensor_nodes_based_on_LoRa_and_LoRaWAN, Jun. 30, 2018 [Feb 22. 203].

[13] ChirpStack. "The ChirpStack Project". Internet: https://www.chirpstack.io/docs/, [Feb. 25, 2023].

[14] ZeroMQ. "Get started". Internet: https://zeromq.org/get-started/, [Mar. 20, 2023].

[15] Control Solutions Minnesota. "Modbus 101 – Introduction to Modbus". Internet: https://www.csimn.com/CSI_pages/Modbus101.html, [May. 11, 2023].

[16] IMST GmbH. "iM282A-L LoRa® Radio Module". Internet: https://wireless-solutions.de/products/lora-solutions-by-imst/radio-modules/im282a-l/, [Feb. 25, 2023].

[17] ElectronicsHub. "Arduino Nano Pinout, Board Layout, Specifications, Pin Description". Internet: https://www.electronicshub.org/arduino-nano-pinout/, Jan 12, 2021 [May 12, 2023].

[18] Linear Technology. "LTC1485 Differential Bus Transceiver". Internet: https://www.analog.com/media/en/technical-documentation/data-sheets/1485fb.pdf, [May 14, 2023].

[19] IONOS Inc. "InfluxDB – explanation, advantages, and first steps". Internet: https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-influxdb/, Sep 28, 2020 [May 11, 2023].

Appendices

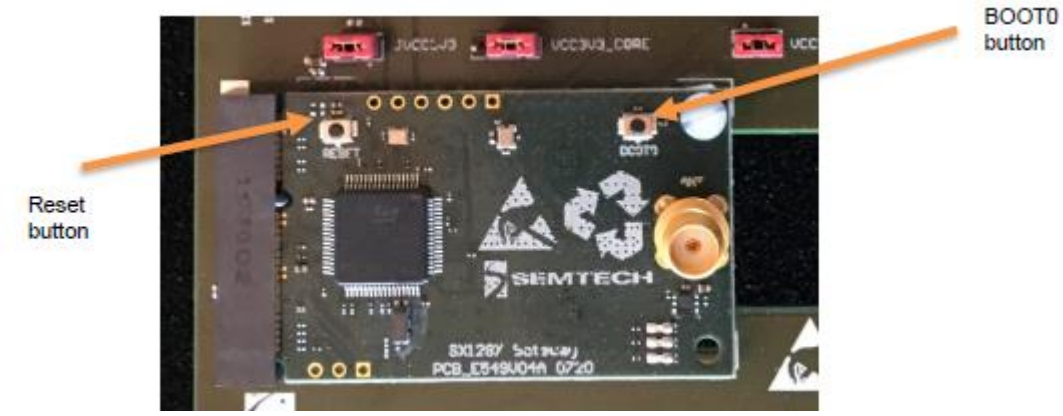
SX1280 Gateway Setup

The steps to setup the ChirpStack Network on the gateway are:

1. Download the Raspbian Buster Lite from https://www.raspberrypi.com/software/.
2. Format the SD card, used the SD Card Formatter software (https://www.sdcard.org/downloads/formatter/) but any will do.
3. Write the downloaded image, used balenaEtcher (https://www.balena.io/etcher) to write the image into the SD card.
4. Once finished, insert the SD card into the Raspberry Pi and connect a HDMI with a monitor and USB keyboard.
5. Once turned on and it is finished booting, the login is "pi" and the password is "raspberry".
6. Enter the following commands:
    a. Sudo apt-get update
    b. Sudo apt-get upgrage
    c. Sudo apt-get dist-upgrade
    d. Sudo rpi-update
    e. Sudo apt install git
    f. Git clone https://github.com/Lora-net/gateway_2g4_hal
    g. cd ~/gateway_2g4_hal/
    h. sudo apt-get install autoconf
    i. git clone https://git.code.sf.net/p/dfu-util/dfu-util
    j. cd dfu-util
    k. ./autogent.sh
    l. Sudo aot-get install libusb-1.0-0-dev
    m. ./configure

n. Make
o. Sudo make install

7. Press the “BOOT0” button of the gateway while pressing the reset button.



8. Back to the Raspberry Pi, the next step is to load the binary into the STM32F446RC MC with the commands:
   a. Cd gateway_2g4_hal/dfu-util
   b. Sudo dfu-util -a 0 -s 0x08000000:leave -t 0 -D ../mcu_bin/rlz_fwm_gtw_2g4_00.02.16.bin
9. Check if the gateway has been recognised with: lsusb.

```
pi@raspberrypi:~/gateway_2g4_hal/dfu-util $
pi@raspberrypi:~/gateway_2g4_hal/dfu-util $ lsusb
Bus 001 Device 009: ID 05c9:5740 Semtech Corp.
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp. SMC9514 Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
pi@raspberrypi:~/gateway_2g4_hal/dfu-util $
```

10. Check what COM device was used to gain access to the gateway (example “ttyACM0”).

```
[58035.761310] cdc_acm 1-1.4:1.0: ttyACM0: USB ACM device
[58367.454148] usb 1-1.4: USB disconnect, device number 8
[58370.050955] usb 1-1.4: new full-speed USB device number 9 using dwc_otg
[58370.200276] usb 1-1.4: New USB device found, idVendor=05c9, idProduct=5740, bcdDevice= 2.00
[58370.200290] usb 1-1.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[58370.200299] usb 1-1.4: Product: GTW 2G4 VPC
[58370.200309] usb 1-1.4: Manufacturer: Semtech
[58370.200318] usb 1-1.4: SerialNumber: 356B39543338
[58370.202773] cdc_acm 1-1.4:1.0: ttyACM0: USB ACM device
pi@raspberrypi:~/gateway_2g4_hal/dfu-util $
```

11. Enter the following commands to complete gateway update:
   a. Cd ~/gateway_2g4_hal/util_boot
   b. Make
   c. ./boot -d /dev/ttyACM0 (use another COM if needed)
   d. Lsusb

```
pi@raspberrypi:~/gateway_2g4_hal/util_boot $ ./boot -d /dev/ttyACM0
Bus 001 Device 010: ID 0483:df11 STMicroelectronics STM Device in DFU Mode
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp. SMC9514 Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
pi@raspberrypi:~/gateway_2g4_hal/util_boot $
```

   e. Cd ~/gateway_2g4_hal/dfu-util
   f. Sudo dfu-util -a 0 -s 0x08000000:leave -t 0 -D ../mcu_bin/rlz_fwm_gtw_2g4_01.00.01.bin
   g. Sudo shutdown now
12. Extract the SD card and then format it again.

13. Download the chirpstack-gateway-os-full-raspberrypi4-20221216152249.rootfs.wic.gz (chirpstack-gateway-os version 4.0.2) from https://artifacts.chirpstack.io/downloads/chirpstack-gateway-os/raspberrypi/raspberrypi4/4.0.2/.
14. Write the downloaded image using etcherBalena.
15. Insert the SD card on the Raspberry Pi, once it is turned on and it is finished booting, the login is “admin” and the password is “admin”.
16. Sudo gateway-config
17. Select “1 Setup LoRa concentrator shield”.
18. Select “14 Semtech – SX1280 LoRa Connect™ (2.4 GHz)”
19. Select “1 ISM2400 (2.4GHz)
20. Select Yes when prompted “Do you want to create the gateway in ChirpStack?”.
21. Select “7 Configure WIFI”, enter SSID and password.
22. Ifconfig, make sure to take note of the gateway’s IP address
23. This step would be used to access the GUI of the ChirpStack network. Go back to your PC, use a web browser to type ip address followed by the port address of 8080 (for example 192.16.1.76:8080).
24. Username is admin, password is admin.

WiMODino Setup

The steps to replace the firmware are as such:

1. Download the WiMOD GlobalLink24 Software in https://wireless-solutions.de/downloadfile/wimod-globallink24-software/.
2. Download and then install the FTDI VCP driver for serial communication via USB from here https://ftdichip.com/drivers/.
3. Download the Arduino library from https://wireless-solutions.de/downloadfile/arduino/ to add to the Arduino IDE.
4. Use a jumper cable to connect GND and GPIO 3 on the WiMODino board.
5. Upload the WiMODino_BtlBridge from WiMOD’s Arduino library to the WiMODino.
6. Close all terminal tools that are connecting to the board.
7. Disconnect and reconnect the board’s USB.
8. Start “WiMOD LoRaWAN EndNode Studio”.
9. Select “File/Firmware Update”.
10. Select the COM port of the board and then continue.
11. Select the iM282A module type and then continue.
12. Quickly remove the jumper cable, if successful the LED will stop to blink. Then continue.
13. Select the installed WiMOD GlobalLink24 firmware to flash and then continue.
14. Close the Firmware Update dialog once finished.
15. Take note of the serial number that is physically stickered on the WiMODino’s iM282A module.
16. Go to https://wireless-solutions.de/support/ to “Open a New Ticket” in order to request for a DEVEUI, make sure to include the device’s serial number.
17. Upload the WiMODino_UARTBridge from WiMOD’s Arduino library to the WiMODino.
18. Open the WiMOD_LoRaWAN_DevTool.
19. Select the COM port of the board and then press the “Open” button.
20. Select known LoRaWAN HC Commands> DevMgmt > Set OpMode Req, insert the value of payload “0x03” and then Execute CMD.

21. Select LoRaWAN> SetDeviceEUI Req, insert the DevEUI on the payload with half the value on the first array and the other half in the other array and then Execute CMD.
22. Now it should be able use the GloabalLinkw24UplinkDownlink_OTAAV2 code. Make sure to change the APPKEY values which can be retrieved when adding the device from Chirpstack. Leave the APPEUI into zeroes.

ModbusExampleSlave

```
#include <SoftwareSerial.h>

#include <SimpleModbusSlaveSoftwareSerial.h>



/* The modbus_update() method updates the holdingRegs register array and checks communication.

  Note:

  The Arduino serial ring buffer is 128 bytes or 64 registers.

  Most of the time you will connect the arduino to a master via serial

  using a MAX485 or similar.

  In a function 3 request the master will attempt to read from your

  slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES

  and two BYTES CRC the master can only request 122 bytes or 61 registers.

  In a function 16 request the master will attempt to write to your

  slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,

  NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write

  118 bytes or 59 registers.

  Using the FTDI USB to Serial converter the maximum bytes you can send is limited

  to its internal buffer which is 60 bytes or 30 unsigned int registers.

  Thus:

  In a function 3 request the master will attempt to read from your

  slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES
```

```
  and two BYTES CRC the master can only request 54 bytes or 27 registers.

  In a function 16 request the master will attempt to write to your
  slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,
  NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
  50 bytes or 25 registers.

  Since it is assumed that you will mostly use the Arduino to connect to a
  master without using a USB to Serial converter the internal buffer is set
  the same as the Arduino Serial ring buffer which is 128 bytes.
*/


// Using the enum instruction allows for an easy method for adding and
// removing registers. Doing it this way saves you #defining the size
// of your slaves register array each time you want to add more registers
// and at a glimpse informs you of your slaves register layout.

//////////////// registers of your slave ///////////////////
enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  PWM_0,
  LED_ON_D7,
  ADC0,
  ADC1,
  ADC2,
  TOTAL_ERRORS,
  // leave this one
  TOTAL_REGS_SIZE = 100
```

```
  // total number of registers for function 3 and 16 share the same register array
};

unsigned int holdingRegs[TOTAL_REGS_SIZE]; // function 3 and 16 register array
//////////////////////////////////////////////////////////////

#define RX          10    // Arduino defined pin (PB0, package pin #5)
#define TX          11    // Arduino defined pin (PB1, package pin #6)
#define RS485_EN     6    // pin to set transmission mode on RS485 chip (PB2, package pin #7)
#define BAUD_RATE    9600  // baud rate for serial communication
#define deviceID     1    // this device address

// SoftwareSerial mySerial(receive pin, transmit pin)
SoftwareSerial rs485(RX, TX);

// Initialise the data to be read
int v1 = 100;
int v2 = 100;
int v3 = 100;
int a1 = 100;
int a2 = 100;
int a3 = 100;
int p1 = 100;
int p2 = 100;
int p3 = 100;
int k1p1 = 100;
int k1p2 = 100;
int k2p1 = 100;
int k2p2 = 100;
int k3p1 = 100;
int k3p2 = 100;
```

```
void setup()
{
 /* parameters(
          SoftwareSerial* comPort
          long baudrate,
          unsigned char ID,
          unsigned char transmit enable pin,
          unsigned int holding registers size)

   The transmit enable pin is used in half duplex communication to activate a MAX485 or similar
   to deactivate this mode use any value < 2 because 0 & 1 is reserved for Rx & Tx
 */
 //configure the modbus parameters
 const int TOTAL_REGS_SIZE = 5000;   //need to check if enum stays at 100 or changed to 5000
 modbus_configure(&rs485, BAUD_RATE, deviceID, RS485_EN, TOTAL_REGS_SIZE);
 pinMode(ADC1, INPUT);
 pinMode(ADC2, INPUT);
 pinMode(7,OUTPUT);
 // assign the holding registers addresses
 holdingRegs[6] = v1;
 holdingRegs[7] = v2;
 holdingRegs[8] = v3;
 holdingRegs[9] = a1;
 holdingRegs[10] = a2;
 holdingRegs[11] = a3;
 holdingRegs[12] = p1;
 holdingRegs[13] = p2;
 holdingRegs[14] = p3;
 holdingRegs[15] = k1p1;
 holdingRegs[16] = k1p2;
```

```
  holdingRegs[17] = k2p1;
  holdingRegs[18] = k2p2;
  holdingRegs[19] = k3p1;
  holdingRegs[20] = k3p2;
  Serial.begin(9600);

}
extern unsigned int errorCount;
void loop()
{
  // modbus_update() is the only method used in loop(). It returns the total error
  // count since the slave started. You don't have to use it but it's useful
  // for fault finding by the modbus master.
  holdingRegs[TOTAL_ERRORS] = modbus_update(holdingRegs);
  if (holdingRegs[1]==1)
  {
    digitalWrite(7,HIGH);
  }
  else
  {
    digitalWrite(7,LOW);
  }
  holdingRegs[ADC0]=analogRead(0);  // check if there are requests to read holding registers.
  delay(100);
  //Serial.println(errorCount);
}
```

ModbusExampleSlave2

```
#include <SoftwareSerial.h>
#include <SimpleModbusSlaveSoftwareSerial.h>
```

```
/* This example code has 9 holding registers. 6 analogue inputs, 1 button, 1 digital output
   and 1 register to indicate errors encountered since started.
   Function 5 (write single coil) is not implemented so I'm using a whole register
   and function 16 to set the onboard Led on the Atmega328P.

   The modbus_update() method updates the holdingRegs register array and checks communication.

   Note:
   The Arduino serial ring buffer is 128 bytes or 64 registers.
   Most of the time you will connect the arduino to a master via serial
   using a MAX485 or similar.

   In a function 3 request the master will attempt to read from your
   slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES
   and two BYTES CRC the master can only request 122 bytes or 61 registers.

   In a function 16 request the master will attempt to write to your
   slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,
   NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
   118 bytes or 59 registers.

   Using the FTDI USB to Serial converter the maximum bytes you can send is limited
   to its internal buffer which is 60 bytes or 30 unsigned int registers.

   Thus:

   In a function 3 request the master will attempt to read from your
   slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES
   and two BYTES CRC the master can only request 54 bytes or 27 registers.

   In a function 16 request the master will attempt to write to your
```

```
  slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,
  NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
  50 bytes or 25 registers.

  Since it is assumed that you will mostly use the Arduino to connect to a
  master without using a USB to Serial converter the internal buffer is set
  the same as the Arduino Serial ring buffer which is 128 bytes.
*/


// Using the enum instruction allows for an easy method for adding and
// removing registers. Doing it this way saves you #defining the size
// of your slaves register array each time you want to add more registers
// and at a glimpse informs you of your slaves register layout.

//////////////// registers of your slave ///////////////////
enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  PWM_0,
  LED_ON_D7,
  ADC0,
  ADC1,
  ADC2,
  TOTAL_ERRORS,
  // leave this one
  TOTAL_REGS_SIZE = 100
  // total number of registers for function 3 and 16 share the same register array
};
```

```
unsigned int holdingRegs[TOTAL_REGS_SIZE]; // function 3 and 16 register array
////////////////////////////////////////////////////////

#define RX          10    // Arduino defined pin (PB0, package pin #5)
#define TX          11    // Arduino defined pin (PB1, package pin #6)
#define RS485_EN     6    // pin to set transmission mode on RS485 chip (PB2, package pin #7)
#define BAUD_RATE    9600  // baud rate for serial communication
#define deviceID     2    // this device address

// SoftwareSerial mySerial(receive pin, transmit pin)
SoftwareSerial rs485(RX, TX);

int vHighGrid = 100, vLowGrid = 100, freqHighGrid = 100;
int freqLowGrid = 100, pfGrid = 100, voltageLoad = 100;
int powerLoad = 100, currentLoad = 100;

void setup()
{
 /* parameters(
          SoftwareSerial* comPort
          long baudrate,
          unsigned char ID,
          unsigned char transmit enable pin,
          unsigned int holding registers size)

   The transmit enable pin is used in half duplex communication to activate a MAX485 or similar
   to deactivate this mode use any value < 2 because 0 & 1 is reserved for Rx & Tx
 */
 const int TOTAL_REGS_SIZE = 5000;
 modbus_configure(&rs485, BAUD_RATE, deviceID, RS485_EN, TOTAL_REGS_SIZE);
 pinMode(ADC1, INPUT);
```

```
  pinMode(ADC2, INPUT);
  pinMode(7,OUTPUT);
  holdingRegs[30] = vHighGrid;
  holdingRegs[31] = vLowGrid;
  holdingRegs[32] = freqHighGrid;
  holdingRegs[33] = freqLowGrid;
  holdingRegs[34] = pfGrid;
  holdingRegs[35] = powerLoad;
  holdingRegs[36] = voltageLoad;
  holdingRegs[37] = currentLoad;
  Serial.begin(9600);

}
extern unsigned int errorCount;
void loop()
{
  // modbus_update() is the only method used in loop(). It returns the total error
  // count since the slave started. You don't have to use it but it's useful
  // for fault finding by the modbus master.
  holdingRegs[TOTAL_ERRORS] = modbus_update(holdingRegs);
  if (holdingRegs[1]==1)
  {
    digitalWrite(7,HIGH);
  }
  else
  {
    digitalWrite(7,LOW);
  }
  holdingRegs[ADC0]=analogRead(0);
  delay(100);
  //Serial.println(errorCount);
```

}

WimodModbusClientV2

```
#include <WiMODGlobalLink24.h>

#include <ArduinoRS485.h> // ArduinoModbus depends on the ArduinoRS485 library

#include <ArduinoModbus.h>

//--------------------------------------------------------------------------

// platform defines

//--------------------------------------------------------------------------

/*

 * Note: This sketch is for Arduino devices with two separate serial interfaces

 * (e.g. DUE). One interface is connected to a PC and one is used for WiMOD.

 *

 * For single serial interface boards (e.g. UNO) it is recommended to disbale

 * the PC / Debug messages

 */


#define WIMOD_IF    SerialWiMOD       // for Mega / Due use: Serial3

#define PC_IF    SerialUSB         // for Mega / Due use: Serial

#define RS845_DEFAULT_DE_PIN D6


// structure for storing the radio parameter set

TWiMODLR_DevMgmt_RadioConfig radioConfigSet;


static TWiMODGlobalLink24_TX_Data txData;


//--------------------------------------------------------------------------

// constant values

//--------------------------------------------------------------------------


/*

 * OTAA Parameters; must be changed by user
```

```
 */

// application  key (64bit)
const unsigned char APPEUI[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }; // CHANGE ME!!!

// application  key (128bit)
const unsigned char APPKEY[] = {0x53, 0x00, 0x6C, 0x1D, 0x8D, 0xF5, 0x46, 0x41, 0xA8,
                                0xB7, 0xFA, 0xE1, 0x9C, 0x1D, 0xCA, 0xE9 };      // CHANGE ME !!!

//------------------------------------------------------------------------------
// section RAM
//------------------------------------------------------------------------------

/*
 * Create in instance of the interface to the WiMOD-GlobalLink24 firmware
 */
WiMODGlobalLink24 wimod(WIMOD_IF);  // use the Arduino Serialx as serial interface

static uint32_t loopCnt = 0;
static uint32_t messageCnt = 0;

//------------------------------------------------------------------------------
// data variables
//------------------------------------------------------------------------------
// for slave id 1
short v1 = 0, v2 = 0, v3 = 0;
short a1 = 0, a2 = 0, a3 = 0;
short p1 = 0, p2 = 0, p3 = 0;
short k1 = 0, k2 = 0, k3 = 0;
short k1P2 = 0, k2P2 = 0, k3P2 = 0;
// for slave id 2
```

```
short vHighGrid = 0, vLowGrid = 0, freqHighGrid = 0;
short freqLowGrid = 0, pfGrid = 0, socSystem = 0;
short powerBatSystem = 0, zeroExportPowerSystem = 0, timeBatSystem = 0;
short voltSystem = 0, docBatSystem = 0, batShutdownSystem = 0;
short voltageLoad = 0, currentSystem = 0, powerLoad = 0;
short batStartSystem = 0, currentLoad = 0;
/*************************************************************************
 * Function for printing out some debug infos via serial interface
 *************************************************************************/
void debugMsg(String msg)
{
   PC_IF.print(msg);  // use default Arduino serial interface
}

void debugMsg(int a)
{
   PC_IF.print(a, DEC);
}

void debugMsgChar(char c)
{
   PC_IF.print(c);
}

void debugMsgHex(int a)
{
   if (a < 0x10) {
      PC_IF.print(F("0"));
   }
   PC_IF.print(a, HEX);
}
```

```
/*****************************************************************************
 * print out a welcome message
 *****************************************************************************/
void printStartMsg()
{
   PC_IF.print(F("================================================\n"));
   PC_IF.print(F("This is FileName: "));
   PC_IF.print(F(__FILE__));
   PC_IF.print(F("\r\n"));
   PC_IF.print(F("Starting...\n"));
   PC_IF.print(F("This program will read data through modbus "));
   PC_IF.print(F("do the OTAA procedure and "));
   PC_IF.print(F("transmit the read messages every 30 sec.\n"));
   PC_IF.print(F("downlink messages will be printed in as hex values.\n"));
   PC_IF.print(F("================================================\n"));
}
/*****************************************************************************
 * rx data callback for U Data
 *****************************************************************************/
void onRxUData(TWiMODLR_HCIMessage& rxMsg) {
   TWiMODGlobalLink24_RX_Data radioRxMsg;
   int i;

   debugMsg("Rx-U-Data Indication received.\n");

   // convert/copy the raw message to RX radio buffer
   if (wimod.convert(rxMsg, &radioRxMsg)) {

   if (radioRxMsg.StatusFormat & LORAWAN_FORMAT_ACK_RECEIVED) {
      // yes, this is an ack
      debugMsg(F("Ack-Packet received."));
```

```
  }
    // print out the received message as hex string
    if (radioRxMsg.Length > 0) {
      // print out the length
      debugMsg(F("Rx-U-Data Message: ["));
      debugMsg(radioRxMsg.Length);
      debugMsg(F("]: "));

      // print out the payload
      for (i = 0; i < radioRxMsg.Length; i++) {
        debugMsgHex(radioRxMsg.Payload[i]);
        debugMsg(F(" "));
      }
      debugMsg(F("\n"));
    }
  }
}
/**************************************************************************
 * rx data callback for U Data
 **************************************************************************/
void onRxCData(TWiMODLR_HCIMessage& rxMsg) {
  TWiMODGlobalLink24_RX_Data radioRxMsg;
  int i;

  debugMsg("Rx-C-Data Indication received.\n");

  // convert/copy the raw message to RX radio buffer
  if (wimod.convert(rxMsg, &radioRxMsg)) {

  if (radioRxMsg.StatusFormat & LORAWAN_FORMAT_ACK_RECEIVED) {
    // yes, this is an ack
```

```
      debugMsg(F("Ack-Packet received."));
    }
      // print out the received message as hex string
      if (radioRxMsg.Length > 0) {
        // print out the length
        debugMsg(F("Rx-C-Data Message: ["));
        debugMsg(radioRxMsg.Length);
        debugMsg(F("]: "));
        // print out the payload
        for (i = 0; i < radioRxMsg.Length; i++) {
          debugMsgHex(radioRxMsg.Payload[i]);
          debugMsg(F(" "));
        }
        debugMsg(F("\n"));
      }
    }
}

/*****************************************************************************
 * rx data callback
 *****************************************************************************/
void onRxAck(TWiMODLR_HCIMessage& rxMsg) {
    TWiMODGlobalLink24_RX_ACK_Data ackData;

    wimod.convert(rxMsg, &ackData);

    debugMsg(F("Received an ACK with status: "));
    debugMsg((int) ackData.StatusFormat);
    debugMsg(F("\n"));
}
/*****************************************************************************
```

```
 * no data indication callback
 ********************************************************************************/
void onNoData(void) {
   debugMsg(F("A No-Data Indication has been received "));
   debugMsg(F("--> (timeout for current (re)transmission)\n"));
}

void onTxUData(TWiMODLR_HCIMessage& rxMsg) {
   TWiMODGlobalLink24_TxIndData sendIndData;

   debugMsg(F("Tx-U-Data Indication: "));

   if (wimod.convert(rxMsg, &sendIndData)) {
      if (   (sendIndData.StatusFormat == LORAWAN_FORMAT_OK )
         || (sendIndData.StatusFormat == LORAWAN_FORMAT_EXT_HCI_OUT_ACTIVE))
      {
         debugMsg(F("TX OK"));
      } else {
         debugMsg(F("TX U-Data failed"));
      }
   }
   debugMsg(F("\n"));
}
/********************************************************************************
 tx indication callback for confirmed data
 ********************************************************************************/
void onTxCData(TWiMODLR_HCIMessage& rxMsg) {
   TWiMODGlobalLink24_TxIndData sendIndData;

   debugMsg(F("Tx-C-Data Indication: "));
```

```
  if (wimod.convert(rxMsg, &sendIndData)) {
    if (   (sendIndData.StatusFormat == LORAWAN_FORMAT_OK )
        || (sendIndData.StatusFormat == LORAWAN_FORMAT_EXT_HCI_OUT_ACTIVE))
    {
      debugMsg(F("TX OK"));

      if (sendIndData.FieldAvailability == LORAWAN_OPT_TX_IND_INFOS_INCL_PKT_CNT) {
        debugMsg(F(" ([re]try #:"));
        debugMsg(sendIndData.NumTxPackets);
        debugMsg(F(")"));
      }
    } else {
      debugMsg(F("TX C-Data failed"));
    }
  }
  debugMsg(F("\n"));
}

/*****************************************************************************
 * Arduino setup function
 *****************************************************************************/
void setup()
{
  // wait for the PC interface to be ready (max 10 sec); usefull for USB
  while (!PC_IF && millis() < 10000 ){}
  // debug interface
  PC_IF.begin(115200);

  SerialUSB.begin(9600);
  while (!SerialUSB);
```

```
  SerialUSB.println("Modbus RTU Client Toggle");

  // start the Modbus RTU client
  if (!ModbusRTUClient.begin(9600)) {
    SerialUSB.println("Failed to start Modbus RTU Client!");
    while (1);
  }

  // init / setup the serial interface connected to WiMOD
  WIMOD_IF.begin(WIMOD_GLOBALLINK24_SERIAL_BAUDRATE);

  // init the communication stack
  wimod.begin();

  printStartMsg();

  // print out some basic info about the module
  wimod.PrintBasicDeviceInfo(PC_IF);

  wimod.DeactivateDevice(); // for multiple starts

 //Change Class A to Class C
  TWiMODGlobalLink24_RadioStackConfig cfg;
     if ( wimod.GetRadioStackConfig(&cfg)) {
        cfg.Options = 0;
        cfg.Options = LORAWAN_STK_OPTION_ADR | LORAWAN_STK_OPTION_DEV_CLASS_C;
         wimod.SetRadioStackConfig(&cfg);
     }

  // register a client callbacks
  wimod.RegisterRxCDataIndicationClient(onRxCData);
```

```
  wimod.RegisterRxUDataIndicationClient(onRxUData);
  wimod.RegisterRxAckIndicationClient(onRxAck);
  wimod.RegisterNoDataIndicationClient(onNoData);
  wimod.RegisterTxCDataIndicationClient(onTxCData);
  wimod.RegisterTxUDataIndicationClient(onTxUData);

  // connect to a LoRaWAN server via an OTAA join
  wimod.ConnectViaOTAA(APPEUI, APPKEY);

}
/*****************************************************************************
 * Arduino loop function
 *****************************************************************************/

void loop()
{
 Serial1.flush();
 delay(30000);
 if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 6, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
 } else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the values.
  v1 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 6 v1: ");
  SerialUSB.println(v1);
 }

 // wait for 1 milisecond
```

```
delay(100);
// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 7, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  v2 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 7 v2: ");
  SerialUSB.println(v2);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 8, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  v3 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 8 v3: ");
  SerialUSB.println(v3);
}
// wait for 1 milisecond
delay(100);
```

```
// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 9, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  a1 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 9 a1: ");
  SerialUSB.println(a1);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 10, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  a2 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 10 a2: ");
  SerialUSB.println(a2);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
```

```
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 11, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  a3 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 11 a3: ");
  SerialUSB.println(a3);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 12, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  p1 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 12 p1: ");
  SerialUSB.println(p1);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 13, 1)) {
```

```
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  p2 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 13 p2: ");
  SerialUSB.println(p2);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 14, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  p3 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 14 p3: ");
  SerialUSB.println(p3);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 15, 2)) {
  SerialUSB.print("failed to read registers! ");
```

```
  SerialUSB.println(ModbusRTUClient.lastError());
 } else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  k1 = ModbusRTUClient.read();
  k1P2 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 15 k1: ");
  SerialUSB.println(k1);
  SerialUSB.print("Read successful from address 16 k1: ");
  SerialUSB.println(k1P2);
 }
 // wait for 1 milisecond
 delay(100);

 // send a Holding registers read request to (slave) id 1, for 1 registers
 if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 17, 2)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
 } else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  k2 = ModbusRTUClient.read();
  k2P2 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 17 k2: ");
  SerialUSB.println(k2);
  SerialUSB.print("Read successful from address 18 k2: ");
  SerialUSB.println(k2P2);
 }
 // wait for 1 milisecond
```

```
delay(100);

// send a Holding registers read request to (slave) id 1, for 1 registers
if (!ModbusRTUClient.requestFrom(1, HOLDING_REGISTERS, 19, 2)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw temperature and the humidity values.
  k3 = ModbusRTUClient.read();
  k3P2 = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 19 k3: ");
  SerialUSB.println(k3);
  SerialUSB.print("Read successful from address 20 k3: ");
  SerialUSB.println(k3P2);
}
// wait for 1 milisecond
delay(100);
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 30, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw voltage(high values.
  vHighGrid = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 30 vHighGrid: ");
  SerialUSB.println(vHighGrid);
}
```

```
// wait for 1 milisecond
delay(100);
// send a Holding registers read request to (slave) id 2, for 1 registers
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 31, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw voltage (low) values.
  vLowGrid = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 31 vLowGrid: ");
  SerialUSB.println(vLowGrid);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 2, for 1 registers
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 32, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw frequency(high) values.
  freqHighGrid = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 32 freqHighGrid: ");
  SerialUSB.println(freqHighGrid);
}
// wait for 1 milisecond
delay(100);
```

```
// send a Holding registers read request to (slave) id 2, for 1 registers
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 33, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw frequency(low) values.
  freqLowGrid = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 33 freqLowGrid: ");
  SerialUSB.println(freqLowGrid);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 2, for 1 registers
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 34, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw power factor from grid values.
  pfGrid = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 34 pfGrid: ");
  SerialUSB.println(pfGrid);
}
// wait for 1 milisecond
delay(100);
```

```
// send a Holding registers read request to (slave) id 2, for 1 registers
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 35, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw power values.
  powerLoad = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 35 powerLoad: ");
  SerialUSB.println(powerLoad);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 2, for 1 registers
if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 36, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
} else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw voltage values.
  voltageLoad = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 36 voltageLoad: ");
  SerialUSB.println(voltageLoad);
}
// wait for 1 milisecond
delay(100);

// send a Holding registers read request to (slave) id 2, for 1 registers
```

```
 if (!ModbusRTUClient.requestFrom(2, HOLDING_REGISTERS, 37, 1)) {
  SerialUSB.print("failed to read registers! ");
  SerialUSB.println(ModbusRTUClient.lastError());
 } else {
  // If the request succeeds, the sensor sends the readings, that are
  // stored in the holding registers. The read() method can be used to
  // get the raw current values.
  currentLoad = ModbusRTUClient.read();
  SerialUSB.print("Read successful from address 37 currentLoad: ");
  SerialUSB.println(currentLoad);
 }
 // wait for 1 milisecond
 delay(100);
 // send out a hello world every 30 sec ( =6* 50*100 ms)
 // (due to duty cycle restrictions 30 sec is recommended
 //if ((loopCnt > 1) && (loopCnt % (6*50)) == 0) {
 String v = String(v1) + "_"+ String(v2) + "_" + String(v3);
 String a = String(a1) +"_"+ String(a2) +"_"+ String(a3);
 String p = String(p1) +"_"+ String(p2)+"_"+ String(p3);
 String k = String(k1) +"_"+ String(k2) +"_"+ String(k3);
 String grid = String(vHighGrid) +"_"+ String(vLowGrid) +"_"+ String(freqHighGrid) +"_"+
String(freqLowGrid)
 + "_" + String(pfGrid);
 String load = String(powerLoad) +"_"+ String(voltageLoad)+"_"+ String(currentLoad);
 String id1 = "1"; // for slave ID 1
 String id2 = "2"; // for slave ID 1
 String id3 = "3"; // for slave ID 2
 String id4 = "4"; // for slave ID 2
 String data1 = id1 +"_"+ String(v) +"_"+ String(a);
 String data2 = id2 +"_"+ String(p)+"_"+ String(k);
 String data3 = id3 +"_"+ String(grid);
```

```
String data4 = id4 +"_"+ String(load);
// send out data
PC_IF.print(F("Sending meter message...\n"));
PC_IF.print(data1 + "\n");
PC_IF.print(data2 + "\n");
PC_IF.print(data3 + "\n");
PC_IF.print(data4);

messageCnt++;

// prepare TX data structure
txData.Port = 0x42;

const int dataSize1 = data1.length() + 1;  // Add 1 for null terminator
char dataCharArray[dataSize1];
strcpy(dataCharArray, data1.c_str());  // Copy string to character array
uint8_t* dataArray = reinterpret_cast<uint8_t*>(dataCharArray);  // Cast to uint8_t array
txData.Length = strlen_P((const char*)dataArray);
strcpy_P((char*)txData.Payload, (const char*)dataArray);
// try to send a message data 1
if (false == wimod.SendUData(&txData)) {
   // an error occurred
   switch (wimod.GetLastResponseStatus()) {
      case GLOBALLINK24_STATUS_DEVICE_NOT_ACTIVATED:
         // activation problem
         // -> try again later
         PC_IF.print(F("-> TX failed: Device is still connecting to server...\r\n"));
         break;
      default:
         PC_IF.print(F("-> TX failed\r\n"));
         break;
```

```
    }

} else {
    PC_IF.print(F("-> TX Done\r\n"));
}
// check for any pending data of the WiMOD
wimod.Process();
delay(30000);

const int dataSize2 = data2.length() + 1;  // Add 1 for null terminator
dataCharArray[dataSize2];
strcpy(dataCharArray, data2.c_str());  // Copy string to character array
dataArray = reinterpret_cast<uint8_t*>(dataCharArray);  // Cast to uint8_t array
txData.Length = strlen_P((const char*)dataArray);
strcpy_P((char*)txData.Payload, (const char*)dataArray);
// try to send a message data 1
if (false == wimod.SendUData(&txData)) {
    // an error occurred
    switch (wimod.GetLastResponseStatus()) {
        case GLOBALLINK24_STATUS_DEVICE_NOT_ACTIVATED:
            // activation problem
            // -> try again later
            PC_IF.print(F("-> TX failed: Device is still connecting to server...\r\n"));
            break;
        default:
            PC_IF.print(F("-> TX failed\r\n"));
            break;
    }

} else {
    PC_IF.print(F("-> TX Done\r\n"));
```

```
}
// check for any pending data of the WiMOD
wimod.Process();
delay(30000);

const int dataSize3 = data3.length() + 1;  // Add 1 for null terminator
dataCharArray[dataSize3];
strcpy(dataCharArray, data3.c_str());  // Copy string to character array
dataArray = reinterpret_cast<uint8_t*>(dataCharArray);  // Cast to uint8_t array
txData.Length = strlen_P((const char*)dataArray);
strcpy_P((char*)txData.Payload, (const char*)dataArray);
// try to send a message data 1
if (false == wimod.SendUData(&txData)) {
   // an error occurred
   switch (wimod.GetLastResponseStatus()) {
      case GLOBALLINK24_STATUS_DEVICE_NOT_ACTIVATED:
         // activation problem
         // -> try again later
         PC_IF.print(F("-> TX failed: Device is still connecting to server...\r\n"));
         break;
      default:
         PC_IF.print(F("-> TX failed\r\n"));
         break;
   }

} else {
   PC_IF.print(F("-> TX Done\r\n"));
}
// check for any pending data of the WiMOD
wimod.Process();
delay(30000);
```

```
  const int dataSize4 = data4.length() + 1;  // Add 1 for null terminator
  dataCharArray[dataSize4];
  strcpy(dataCharArray, data4.c_str());  // Copy string to character array
  dataArray = reinterpret_cast<uint8_t*>(dataCharArray);  // Cast to uint8_t array
  txData.Length = strlen_P((const char*)dataArray);
  strcpy_P((char*)txData.Payload, (const char*)dataArray);
  // try to send a message data 1
  if (false == wimod.SendUData(&txData)) {
     // an error occurred
     switch (wimod.GetLastResponseStatus()) {
        case GLOBALLINK24_STATUS_DEVICE_NOT_ACTIVATED:
           // activation problem
           // -> try again later
           PC_IF.print(F("-> TX failed: Device is still connecting to server...\r\n"));
           break;
        default:
           PC_IF.print(F("-> TX failed\r\n"));
           break;
     }

  } else {
     PC_IF.print(F("-> TX Done\r\n"));
  }

  // check for any pending data of the WiMOD
  wimod.Process();

  loopCnt++;
}
```

Codec (ChirpStack JavaScript code)

```
// Decode uplink function.

// Source code:
https://www.youtube.com/watch?v=aliHXCR564k&ab_channel=AnhQu%C3%A2nT%E1%BB%91ng

// Input is an object with the following fields:

// - bytes = Byte array containing the uplink payload, e.g. [255, 230, 255, 0]

// - fPort = Uplink fPort.

// - variables = Object containing the configured device variables.

//

// Output must be an object with the following fields:

// - data = Object representing the decoded payload.


function hex_to_ascii(tohex) {
   var hex = tohex.toString();
   var str = '';
   for (var n = 0; n < hex.length; n += 2) {
      str += String.fromCharCode(parseInt(hex.substr(n, 2), 16));
   }
   return str;
}


function toHexString(bytes) {
   return bytes.map(function(byte) {
      return ("00" + (byte & 0xFF).toString(16)).slice(-2)
   }).join('')
}



function decodeUplink(input ) {
  var tohex = toHexString(input.bytes);
  var toascii = hex_to_ascii(tohex);
  const dataArray = toascii.split("_"); // split the string into an array of substrings
```

```
// convert the substrings to an integer and store it in a variable
const idVariable = parseInt(dataArray[0]);
const firstVariable = parseInt(dataArray[1]);
const secondVariable = parseInt(dataArray[2]);
const thirdVariable = parseInt(dataArray[3]);
const fourthVariable = parseInt(dataArray[4]);
const fifthVariable = parseInt(dataArray[5]);
const sixthVariable = parseInt(dataArray[6]);

if (idVariable == 1) {
  return {
    data: {
      v1: firstVariable,
      v2: secondVariable,
      v3: thirdVariable,
      a1: fourthVariable,
      a2: fifthVariable,
      a3: sixthVariable
    }
  };
}
else if (idVariable == 2) {
  return {
    data: {
      p1: firstVariable,
      p2: secondVariable,
      p3: thirdVariable,
      k1: fourthVariable,
      k2: fifthVariable,
      k3: sixthVariable
    }
```

```
    };
  }
  else if (idVariable == 3) {
    return {
      data: {
        vHighGrid: firstVariable,
        vLowGrid: secondVariable,
        freqHighGrid: thirdVariable,
        freqLowGrid: fourthVariable,
        pfGrid: fifthVariable
      }
    };
  }
  else if (idVariable == 4) {
    return {
      data: {
        powerLoad: firstVariable,
        voltageLoad: secondVariable,
        currentLoad: thirdVariable
      }
    };
  }
}

// Encode downlink function.
//
// Input is an object with the following fields:
// - data = Object representing the payload that must be encoded.
// - variables = Object containing the configured device variables.
//
// Output must be an object with the following fields:
```

```
// - bytes = Byte array containing the downlink payload.
function encodeDownlink(input) {
  return {
    bytes: [225, 230, 255, 0]
  };
}
```

chirpstackDownlink.py

```
# Source: https://www.chirpstack.io/docs/chirpstack/api/python-examples.html?highlight=Downlink#enqueue-downlink
import os
import sys
from time import sleep

import grpc
from chirpstack_api import api
from influxdb import InfluxDBClient
import influxdb_client
from influxdb_client.client.write_api import SYNCHRONOUS
# Configuration.

# This must point to the API interface.
server = "192.168.1.76:8080"

# The DevEUI for which you want to enqueue the downlink.
dev_eui1 = "70B3D58FF0040150"

# The API token (retrieved using the web-interface).
api_token1 = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJjaG" \
             "lycHN0YWNrIiwiaXNzIjoiY2hpcnBzdGFjayIsInN1YiI6IjEzN" \
             "DMzNWVkLWMwYTMtNDQ4Ny1iZjRiLTZiYTE1NTU3ZjllYyIsInR" \
             "5cCI6ImtleSJ9.mBNIpfeD7kQ1cVU9pmCUzNtQAUKjeopbMihYL_EjUm4"
```

```
bucket = "chirpstack"

org = "Project"

token =
"hcNv6G9csIfAvSpQB1Rt3hAnUOjf4CJttLAlIoctLi_zqvz_jKWhgQgbMEYXFlxSLyeiwOXaFhLSOnC1LKXFg
g=="

# Store the URL of your InfluxDB instance

url="http://localhost:8086"

if __name__ == "__main__":

 # Connect without using TLS.

 channel = grpc.insecure_channel(server)


 # Device-queue API client.

 client = api.DeviceServiceStub(channel)


 # Define the API key meta-data.

 auth_token1 = [("authorization", "Bearer %s" % api_token1)]


 # Construct request for device 1 (smart meter).

 req = api.EnqueueDeviceQueueItemRequest()

 req.queue_item.confirmed = False

 req.queue_item.data = bytes([0x01, 0x02, 0x03])

 req.queue_item.dev_eui = dev_eui1

 req.queue_item.f_port = 10


 # connect to InfluxDB

 influxClient = influxdb_client.InfluxDBClient(

    url=url,

    token=token,

    org=org

 )

 query_api = influxClient.query_api()
```

```
# Construct request for device 2 (inverter).
#req = api.EnqueueDeviceQueueItemRequest()
#req.queue_item.confirmed = False
#req.queue_item.data = bytes([0x01, 0x02, 0x03])
#req.queue_item.dev_eui = dev_eui2
#req.queue_item.f_port = 10

while True:
    # send downlink transmission to smart meter
    resp = client.Enqueue(req, metadata=auth_token1)
    # Print the downlink id
    print(resp.id)

    # create continuous query for influx
    query = 'from(bucket:"chirpstack") ' \
            '|> range(start: -30s)'
    result = query_api.query(org=org, query=query)
    results = []
    for table in result:
        for record in table.records:
            results.append((record.get_field(), record.get_value()))

    print(results)

    # send downlink transmission to inverter
    #resp = client.Enqueue(req, metadata=auth_token1)
    # Print the downlink id
    #print(resp.id)
    sleep(30)
```