TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

IE70LT
Muhammad Faizan Aziz Khan IVEM165530

# LORA BASED SMART AGRICULTURE SYSTEM

Master's thesis

Supervisor: Dr. Paul Annus

PhD
Senior Research
Scientist

Co-Supervisor: Dr. Tauseef Ahmed

PhD
Lecturer

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

IE70LT
Muhammad Faizan Aziz Khan IVEM165530

# LORA SÜSTEEMIL PÕHINEV PÕLLUMAJANDUSSÜSTEEM

Magistritöö

Juhendaja: Dr. Paul Annus

PhD
Vanemteadur

Kaasjuhataja: Dr. Tauseef Ahmed

PhD
Lektor

Tallinn 2018

# Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Muhammad Faizan Aziz Khan

13.05.2017

# Abstract

Advancements in low power and low-cost computation and communication technology have brought a revolution in remote sensing and monitoring applications. The Internet of Things paradigm promises an ecosystem of connected devices spread across a vast variety of application domains. This thesis deals with the practical aspects of engineering such a system of connected devices for monitoring agricultural systems.

Currently, there are many competing standards and technologies trying to take a hold of IoT, especially the area of remote sensing and communication technology. LoRa is one of these technologies gaining popularity in the application of Wireless Sensor Networks (WSNs). The ability of LoRa to establish communication links over long distances with relatively simple nodes, minimal infrastructure, low power requirements and utilization of license-free ISM bands give it a considerable edge of its competitors. Although a lot of research work has been done about the efficacy of LoRa for low power wireless sensor networks, there are still gaps in the literature about the practical aspects of design and implementation of such systems. This work focuses on problem of implementing a localized sensor network for a smart agriculture system using LoRa as the main communication technology. A system for monitoring temperature, pressure and humidity is implemented using low cost solutions available in the market and results are presented.

This thesis is written in English and is 49 pages long, including 5 chapters, 36 figures and 1 table.

# Annotatsioon

# LoRa süsteemil põhinev Põllumajandussüsteem

Madala võimsusega ja maksumusega arvutustehnika ja kommunikatsioonitehnoloogia progress on teinud kaugseire- ja seirerakenduste valdkonnas revolutsiooni. Asjade Interneti paradigma kirjeldab ühendatud seadmete ökosüsteemi, mis on levinud paljude rakenduste domeenide hulgas. Käesolev tees kirjeldab inseneritöö praktilisi aspekte nagu põllumajandussüsteemide seireks mõeldud ühendatud seadmete süsteem.

Tänapäeval on palju konkureerivaid standardeid ja tehnoloogiad, mis tegutsevad Asjade Interneti valdkonnas, eriti kaugseire ja kommunikatsioonitehnoloogia valdkonnas. LoRa on üks nendest tehnoloogiatest, mis saab veelgi populaarsemaks Traadita Andurite Võrgustiku rakendamise valdkonnas. LoRa on võimeline looma sideühendusi pikkade vahemaade üle suhteliselt minimaalse infrastruktuuriga ja madalate energiatarbenõuetega, ja lisaks, litsentsivaba ISM-ide kasutamine annab sellele suurepärast konkurentsivõimet. Vaatamata sellele, et on juba tehtud palju uuringuid LoRa efektiivsuse väikese võimsusega traadita andurite võrgustiku seoses, kirjanduses on ikka veel puudusi selliste süsteemide disainimise ja rakendamise praktiliste aspektide kohta. See töö keskendub traadita andurite võrgustiku väikeste põllumajandussüsteemile rakendamisele, kasutades LoRa kui peamist kommunikatsioonitehnoloogiat. Süsteem on mõeldud temperatuuri, rõhu ja niiskuse seireks ja seda rakendatakse kasutades turul saadaval olevaid madala maksumusega lahendusi ning tulemused on esitatud.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 49 leheküljel, 5 peatükki, 36 joonist, 1 tabelit.

# List of abbreviations and terms

| | |
|---|---|
| BLE | Bluetooth Low Energy |
| IoT | Internet of Things |
| LoRa | Long Range Radio Protocol |
| LPWSN | Low Power Wireless Sensor Network |
| LPWAN | Low Power Wide Area Network |
| WSN | Wireless Sensor Node/Wireless Sensor Node |
| Wi-Fi | Wireless Fidelity |
| RTOS | Real Time Operating System |
| DVFS | Dynamic Voltage and Frequency Scaling |
| HTML | Hypertext Mark-up Language |
| TOA | Time on Air |
| SNR | Signal to Noise Ratio |
| DSSS | Direct Sequence Spread Spectrum |
| RF | Radio Frequency |
| OSI | Open System Interconnection |
| ISM | Industrial Scientific Medical |
| MCU | Microcontroller Unit |
| GPS | Global Positioning System |
| CSS | Chirp Spread Spectrum |
| FSK | Frequency Shift Key |
| UART | Universal Asynchronous Receiver Transmitter |
| I2C | Inter Integrated Circuit |
| GPIO | General purpose Input Output |
| FIFO | FIRST-in First-out |

# Table of Contents

# List of figures

# List of tables

# 1 Chapter 1

This chapter covers the motivation and goals of the project with a transitory introduction to the Internet of Things (IoT), Low Power Wide Area Networks (LPWAN), the importance of Wireless Sensor Network (WSN) in different sectors, particularly in agriculture system.

## 1.1 Introduction

The IoT is rapidly growing area of development for the scientific community. IoT evolutionary process had made way for wireless, multipurpose and low power devices by obsoleting wired communication, single function and high-power consumption devices. The demand for IoT is increasing day by day in every sector of life as companies are launching new devices in the market. The information collected and processed by the devices can be transferred to the internet for human analysing, understanding and feedback. Facts and figures claim that by 2020 the number of IoT devices will rise roughly up to 26 billion [1]. As seen from the Gartner Hype Cycle [2] in Figure 1 emphasis on three emerging technologies one of them is IoT.

With the advancement in computing technology in the recent decades, compact, low power and inexpensive microprocessors have pushed the interest in IoT. This resulted in a need for robust, low power and flexible communication protocols to suit the needs of IoT applications in both urban and rural environments. IoT devices have to be able to operate on battery power for long periods of time. A network of such devices can be used to monitor and report various physical and environmental variables like pressure, humidity, temperature, sound, motion etc. These so-called ‘Wireless Sensor Networks’ or WSNs can collect data from a wide area and report to a central sink where it can be uploaded to the cloud and analysed. Although, the existing mobile networks can support applications to some extent the increasing demand of IoT end users is already taxing the current infrastructure [3]. In a few years, there could be as much as a billion WSNs

connected to the internet simultaneously [3]-[4]. Evidently, there is a need for an infrastructure-less communication protocol to suit these needs.

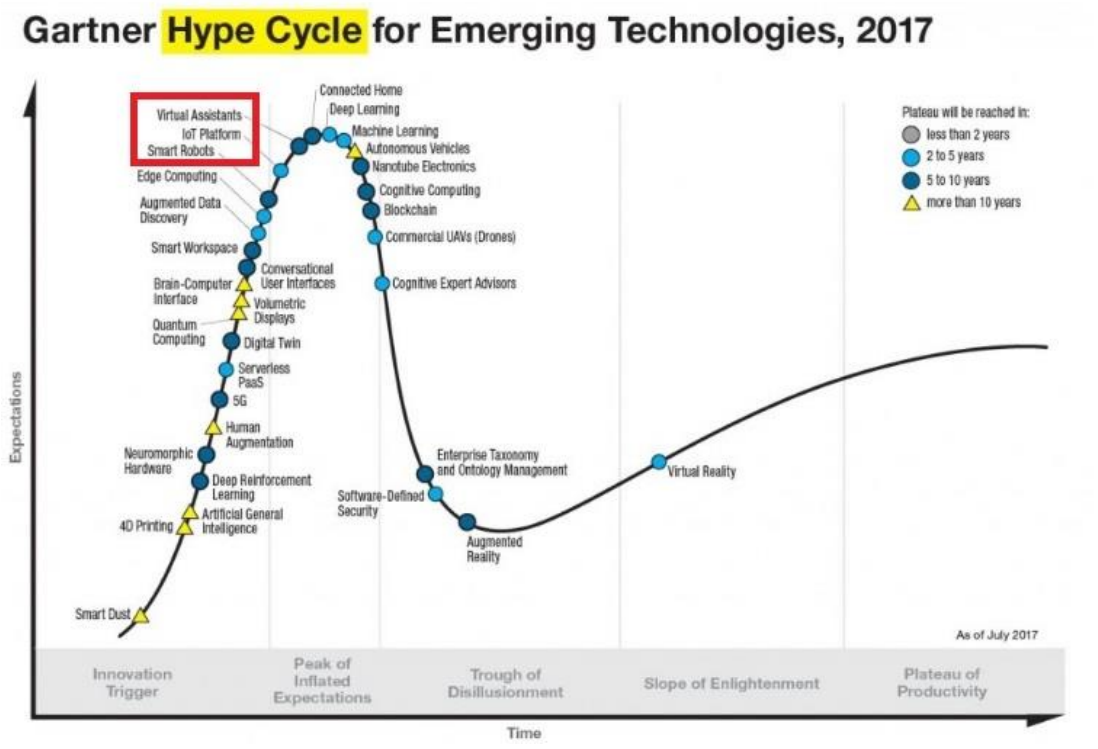


Figure 1. Gartner Hype Cycle 2017 [5].

## 1.2 WSN Background

Wireless sensor networks are a set of infrastructure-less nodes for monitoring and collection of data about their environment. Usually, this data consists of variables like temperature, pressure, humidity and so on. Broadly speaking, WSNs are not only limited to data collection and may include actuators to control their environment as well [6]. Individual nodes collect and forward data, periodically, to one or more sinks [7]. The sink processes and forwards the data to be used locally or uploaded to the cloud through a gateway as shown in Figure 2. A single sink WSN scenario is simpler for applications with a limited number of nodes but lacks scalability. A generic WSN usually has multiple sinks with the flexibility to add mode sinks to increase network capacity. Other factors affecting the choice of a number of sinks include minimum delay, maximum throughput, number of hops, and protocol complexity [7].

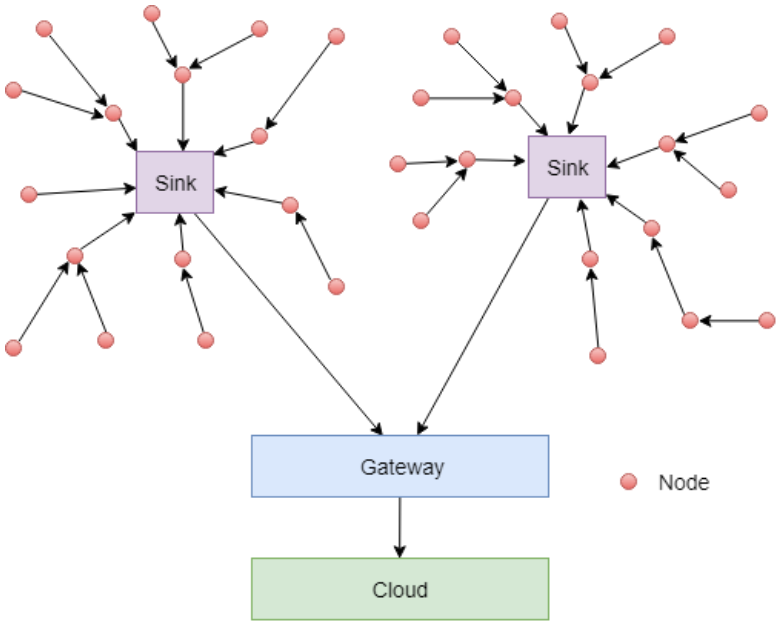


Figure 2. Wireless Sensor Network Connected to Cloud.

Wireless Sensor Network (WSN) is like a sensory nervous system that connects the real world to the digital world. A WSN is a network of small wireless electronic nodes also known as sensor nodes. The purpose of WSN is to collect data from the environment and pass it to cloud services or databases to monitor, store and make a decision on sensed data. WSN is projected as an economically feasible model and a promising technology because of its ability to offer a variety of service areas, such as weather monitoring, security, surveillance, agriculture system and home automation [8].

Low Power Wide Area Network (LPWAN) is a type of communication in WSN designed to allow long-range communication at low bit rate among connected objects. Leading standards like Bluetooth Low Energy (BLE), ZigBee, Z-wave are ruling in IoT [9].

### 1.2.1 Brief History

Although the idea of remote sensing is not new, research interest in WSNs started in the 80s [10], the availability of cheap and energy efficient components in the early 21$^{st}$ century made WSNs feasible for real-world applications.

One of the first real-world applications that drove the development for WSNs was military surveillance in war zones [10]. Later, they were extended to monitoring environmental conditions in the industry infrastructure, traffic regulation, medical and health, industrial automation and other similar applications. Usually, such applications use a distributed network of independent sensor nodes. United States Defence Advanced Research Projects

Agency, DARPA, developed the initial WSNs in the 80s for the US military [11]. The Distributed Sensor Network (DSN) program was carried out to develop low cost, low power and autonomous sensor nodes that collected sensor data and routed it through the nearest neighbour nodes to whichever node needed the data. The biggest challenge in developing DSN nodes was that the technology at that time wasn't quite ready. Hence, the DSN nodes were bulky and power hungry which limited potential applications [10].

Since then, the technology has caught up to the requirements of WSNs. Advancements in low power microprocessors, sensor technology, and power electronics have led to the development of sensor nodes that can work for months on a single battery. Recently, developments in energy harvesting technology have led to a new wave of possibilities. Energy harvesting WSN nodes can potentially work indefinitely without needing any sort of service.

### 1.2.2 Sensor Nodes

Typically, a WSN node consists of a power source, a microcontroller for processing, analog and/or digital sensors and a suitable wireless transceiver as shown in Figure 3.

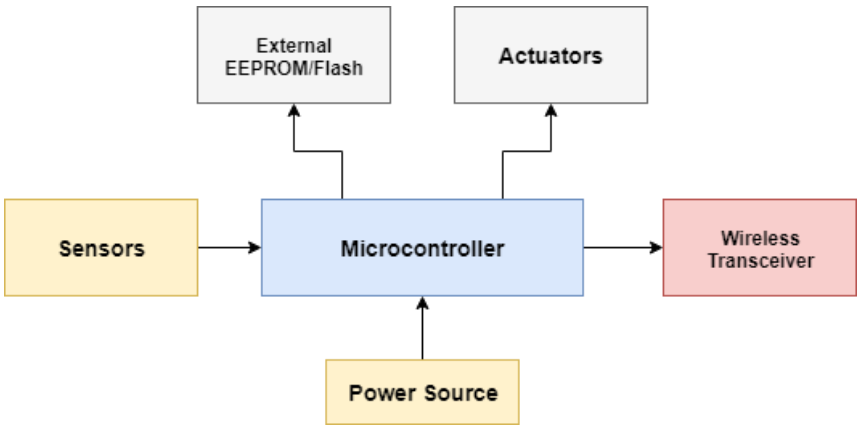


Figure 3. Typical WSN Node.

The choice of sensors depends upon the data variables of interest in the environment. These sensors can be either analog or digital. Some analogue sensors may require some signal conditioning circuitry before they can be fed to the microcontroller. The microcontroller handles all the computational tasks of a WSN node. Factors that affect the choice of microcontroller include mixed signal capability, cost, I/O, memory, and power efficiency. These constraints limit the choice of microcontrollers to low-power embedded processors families that are specifically targeted for low power applications.

These processors usually belong to the small MHz area which limits the computational complexity that can be handled by a single node. In commercial applications, these devices usually run proprietary Real-Time Operating Systems (RTOS) specifically designed for low power applications. Advanced techniques for optimizing power usage like sleep modes and dynamic voltage and frequency scaling (DVFS) are incorporated in these operating systems. Some nodes may have additional elements like permanent external memory for data logging and actuators like switches, valves, motors etc.

The most important part of any WSN node is the wireless transceiver. The choice of protocol depends on factors like power consumption, reliability, size, mobility, privacy and security etc [7]. Transceiver radios with long-range, low power and low data rate are used along with a suitable protocol for any specific application. The choice of transceivers depends upon factors like noise immunity, spectral efficiency, fading, interference robustness, cost etc [7]. Radio transceivers are usually the most power-hungry components of WSN nodes hence spectral efficiency and energy efficiency are the major factors affecting the choice of transceivers.

Some WSN nodes also have positioning and ranging capabilities to geotag the collected data. Knowledge about the location of data collection point is vital in the analysis of collected data for a majority of WSN applications. It's most intensive and time-consuming task to pre-configure the position of each node within the environment. Satellite-based positioning systems like GPS are a common choice for addressing these issues.

### 1.2.3 Networking Structure and Topology

The choice of networking topology in WSNs affects the reliability, complexity, scalability, and efficiency of the network. Moreover, the optimum location of sensor nodes also depends on the networking topology used [12]. Following are common topologies usually employed in WSNs.

### 1. Star Topology-Single Hop

As the name indicates, in this topology, every node is one hop away from the sink. This is the simplest WSN topology as shown in Figure 4. Every node is only an endpoint hence there is no need to accommodate any sort of forwarding capability in sensor nodes. This greatly simplifies the design and minimizes the computational requirements of each node. The most significant drawback of this topology is poor scalability and robustness. Moreover, this topology is not suitable for covering a wide area since any nodes distant nodes will suffer from a poor link and signal quality to the sink.

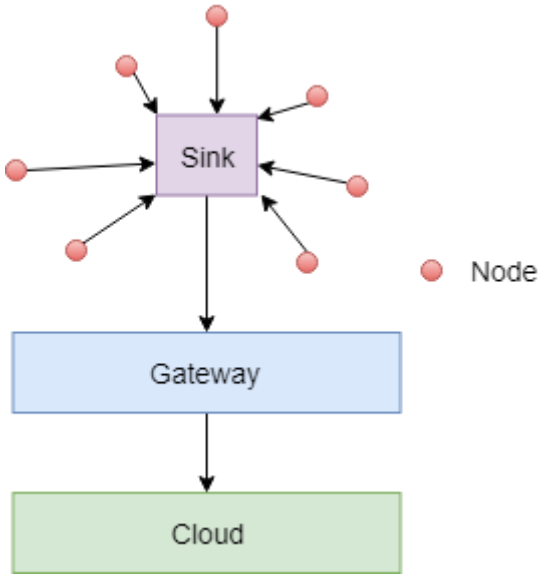


Figure 4. Single Hop Star.

### 2. Mesh Topology-Multi Hop

This topology covers the shortcomings of single hop start topology by adding routing capability to nodes as shown in Figure 5. Hence, sensor nodes are capable of taking the role of both endpoints and routers for other endpoints. The signal is transmitted from one node to the other until it reaches the sink. The path taken by the signal depends on the specific routing protocol implemented. This adds complexity to the design of sensor nodes. Moreover, router nodes consume more power as compared to end nodes.

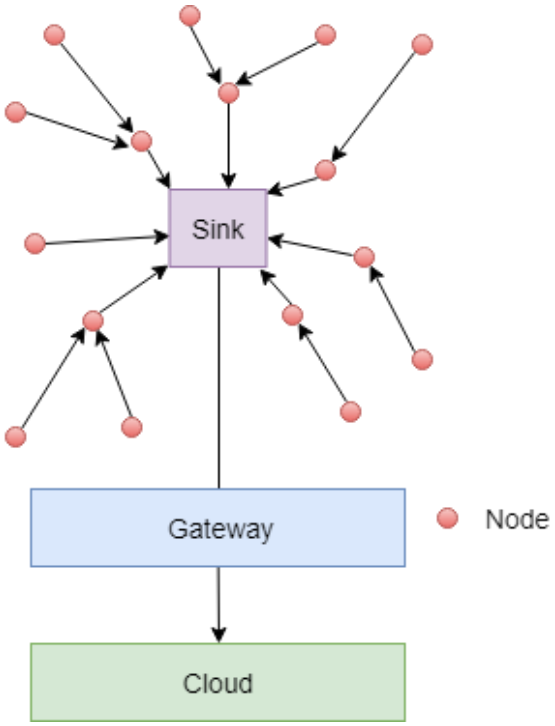


Figure 5. Multi-hop Mesh Topology.

### 3. Hierarchical Clusters

This is one of the most common network topology suitable for large WSNs. The network is divided into groups called clusters. These clusters usually have a special node called cluster-head to which all the other sensor nodes report. For relatively larger networks, there can be multiple levels of clusters within clusters. The main advantage of this hierarchical scheme is that network management can be localized in each cluster which results in simpler nodes and routing protocols. Cluster-heads can use a different, more powerful transmission link, as compared to sensor nodes, in connection with other cluster heads and sinks [7].

### 1.2.4 Routing Protocols

One of the most important objective in WSNs is optimizing power efficiency. In multi-hop networking topologies, routing protocols play a major role in optimizing efficiency and delay associated with the network. The key objective of routing protocols is to ensure the selection of a best possible route for data. Routing is an optimization problem in its essence and some of the key parameters for this problem are a number of hops, delay, and load balancing between the nodes.

### 1.2.5 Application Areas

One of the first application of WSNs was in military surveillance. This sparked the interest in research and development of WSNs through the years. Recent advances in technology have opened new application areas like environmental monitoring, flood detection, smart grid, home automation, vehicle tracking, traffic flow monitoring etc. This thesis focuses on the application of WSN in the field of automated agricultural monitoring. Agricultural monitoring systems collect environmental data for crop fields for optimization of growth and yield [13]. WSN nodes can collect a variety of data affecting plant growth conditions like humidity, temperature, pressure, soil moisture content etc. This can not only reduce the time and effort for monitoring the conditions of a farm but also eliminate the likelihood of measurement errors and sporadic readings [14]. Automated crop monitoring systems can help identify diseases and improve yield for large fields [15]. Agricultural monitoring systems have shown significant promise in precision agriculture [16]–[21].

## 1.3 Motivation

Agriculture system has been around for thousands of years allowing humankind to expand and construct permanent settlements. Environmental factors such as water, temperature, pressure, moisture, rain and many more effect plant growth significantly. Agricultural environments such as open fields and greenhouse allow farmers to produce plants with an emphasis on agriculture yield and productivity. Modern day technology empowers mankind to grow plants in environments previously not suitable for task [22].

The development of wireless communication technology in the last decade has made wireless communication protocols exclusive in the domain of sensor networks. Existing trends have encouraged the use and implementation of many radio based protocols due to fact that short-range the radio transmission is inexpensive, secure and easily available. Short range standards like ZigBee, Bluetooth and Wi-Fi have been on the top of list for short-range communication [22]. On the other hand, long-range communication is under development process.

Therefore, the objective of the thesis is to design and implement a LoRa based Wireless Sensor Network for an agriculture system capable of intelligently monitoring parameters

affecting production and quality of plants. Figure 6 describes the basic concept of smart agriculture system [23].

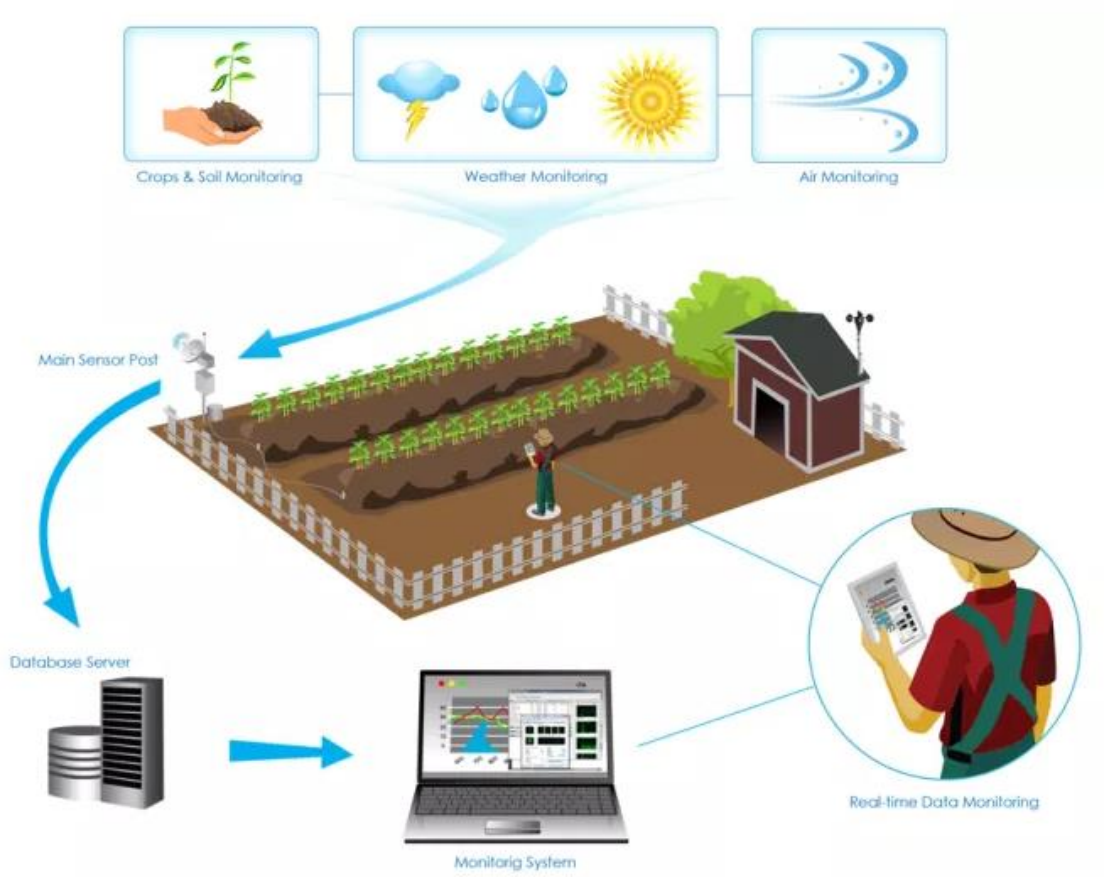


Figure 6. Smart Agriculture System [23].

## 1.4 Problem statement

The problem statement for this thesis work is to design and implement a low-cost LoRa based smart agriculture system and evaluate it. Most of the research done in this area has been focussed on short-range communication. Less amount of work has been done in terms of practical implementation of LoRa as a wireless sensor network. This work will give clear overview and understanding of design and application of LoRa in agriculture field [24].

## 1.5 Objectives and Goals

The primary objective of this thesis involves designing, implementation and analysing long range communication protocol in agriculture system which should be capable of:

- Measurement of factors affecting production and quality of crops.
- Sending and receiving the data over long range.
- Suitable for both environments indoor and outdoor.
- Creating the model hardware and software architecture which can be used for development.
- Testing, analysing and comparison of range with the industry standard.
- Easy to implement, low cost, power efficient and environment-friendly.

## 1.6 Significance of work

Internet of Things (IoT) has grown-up enormously over the period due to variety and capability of their modern and real-world applications in areas like smart homes, environmental applications and e-health. Wireless sensor network (WSN) is like the eyes and ears of the Internet of Things (IoT). Typically, sensor nodes use low power short range communication protocols like ZigBee, Bluetooth Low Energy (BLE) and Infrared Transmission which are not effective for long-range communication. By implementing low power long range communication protocols like Sigfox and LoRa we can cover more distance.

## 1.7 Report structure and work frame

The overall thesis report consists of five chapters. Chapter 1 covers general introduction, motivation, and goals, followed by Chapter 2 focus on literature review and methodology. Chapter 3 provides a detailed description of devices, hardware, and software requirements to execute the project. Chapter 4 covers implementation, design and deployment. Chapter 5 covers results and discussion and is based on all previous chapters represents a conclusion, future work, and summary of work. The whole task was divided into 4 sub-tasks mentioned in Table 1 to meet the deadline.

Table 1. Time scheduling for tasks.

| **NR.** | **Description of Task** | **Time frame** |
|---|---|---|
| 1 | Research on background and literature work | 2 weeks |
| 2 | Choosing and learning about hardware | 2 weeks |
| 3 | Implementation of model | 2 weeks |
| 4 | Writing report | 3 weeks |

# 2 Chapter 2

## 2.1 System Overview

There are three main components of the system, as shown in Figure 7. The endpoint is the sensor node that collects environment data and forwards it to the gateway. The gateway collects and logs all the forwarded data and provides the logged data to the web interface. The user can interact with the data through a browser. The link between endpoints and gateway is LoRa based whereas the web interface is accessed through Wi-Fi.


Figure 7. System Overview.

### 2.1.1 Endpoint

The endpoint consists of a low power, a mixed signal microcontroller connected to an array of sensors collecting pressure, temperature and humidity data. This data is periodically sent to the gateway over LoRa through a transceiver module.

### 2.1.2 Gateway

The gateway consists of a single board computer running embedded Linux. The gateway has a LoRa transceiver to receive data from endpoints and Wi-Fi module for web connectivity. It runs an apache server to host a simple web interface that displays the data received from various endpoints. The gateway is supposed to run from a reliable power source unlike the endpoints since it consumes considerably more power.

### 2.1.3 Web Interface

The web interface is a simple HTML page accessible through any browser. It can be accessed by any personal devices connected to the same network.

## 2.2 LoRa Introduction

LoRa is short for “Long Range”, is a wireless communication technology developed specifically for long range and low power communication applications. It used spread spectrum technique and chirp modulation to transmit data over a wide range of frequencies from 137 MHz to 1020 MHz therefore, quite a few license-free ISM bands can be used for LoRa communication (169 MHz, 433 MHz, 868 MHz and 915 MHz) [25]. LoRa sacrifices data rate for range hence it is only suitable for applications that need to transmit small amounts of data periodically. This makes LoRa extremely useful for WSN applications. LoRa is the first low-cost commercial application for chirp spread spectrum [26].

## 2.3 LoRa Network Architecture

LoRa is promoted as an infrastructure solution for Internet of Things by Samtech and the LoRa Alliance. It consists of end devices which communicate with a gateway through a single hop. The gateway acts as a bridge between endpoints and the internet/cloud. Gateways log and relay the data between end devices and cloud servers [27]. Each gateway in a LoRa network makes a single-hop star network of endpoints around it. Similarly, all gateways in the infrastructure are connected to the cloud. This makes LoRa network a” star-of-stars” topology as shown in Figure 8.

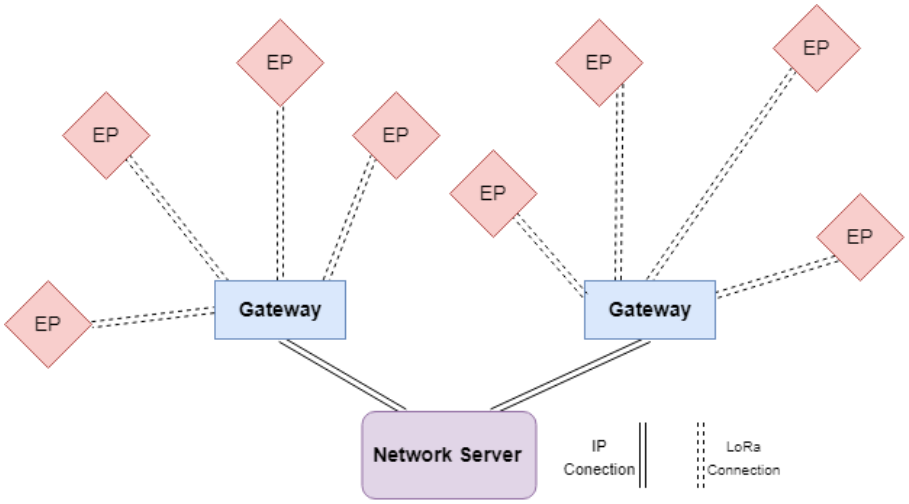


Figure 8. Network Topology.

## 2.4 LoRa Modulation Scheme

LoRa utilizes chirp spread-spectrum modulation [28] scheme for achieving long-range with low power. The actual scheme used by Samtech in LoRa PHY is closed source but it's a derivative of chirp spread-spectrum modulation. LoRa uses orthogonal spreading factors to implement a variable data rate for range and power for various application-specific needs. Different combinations of spreading factor, coding rate and bandwidth results in various modes which can be utilized to achieve the required range and data rate.

The following Figure 9 shows the time on air (TOA) for various combinations of spreading factors, coding rates, and bandwidths.

Range

Throughput

| | | | time on air in second for payload size of | | | | | |
|---|---|---|---|---|---|---|---|---|
| BW | CR | SF | 5 bytes | 55 bytes | 105 bytes | 155 Bytes | 205 Bytes | 255 Bytes |
| 125 | 4/5 | 12 | 0.95846 | 2.59686 | 4.23526 | 5.87366 | 7.51206 | 9.15046 |
| 250 | 4/5 | 12 | 0.47923 | 1.21651 | 1.87187 | 2.52723 | 3.26451 | 3.91987 |
| 125 | 4/5 | 10 | 0.28058 | 0.69018 | 1.09978 | 1.50938 | 1.91898 | 2.32858 |
| 500 | 4/5 | 12 | 0.23962 | 0.60826 | 0.93594 | 1.26362 | 1.63226 | 1.95994 |
| 250 | 4/5 | 10 | 0.14029 | 0.34509 | 0.54989 | 0.75469 | 0.95949 | 1.16429 |
| 500 | 4/5 | 11 | 0.11981 | 0.30413 | 0.50893 | 0.69325 | 0.87757 | 1.06189 |
| 250 | 4/5 | 9 | 0.07014 | 0.18278 | 0.29542 | 0.40806 | 0.5207 | 0.63334 |
| 500 | 4/5 | 9 | 0.03507 | 0.09139 | 0.14771 | 0.20403 | 0.26035 | 0.31667 |
| 500 | 4/5 | 8 | 0.01754 | 0.05082 | 0.08154 | 0.11482 | 0.14554 | 0.17882 |
| 500 | 4/5 | 7 | 0.00877 | 0.02797 | 0.04589 | 0.06381 | 0.08301 | 0.10093 |

Figure 9. TOA for Various Combinations of LoRa Settings [29].

Following is a recap of some ideas needed to understand the LoRa modulation scheme.

### 2.4.1 Shannon-Hartley Theorem

In information theory, the Shannon-Hartley theorem dictates the maximum achievable transmission rate of a communication channel [30]-[31]. For given channel with specified bandwidth and noise, the channel capacity of the communication link is given as:

$$C = B * log_2\left(1 + {}^{S}/_{N}\right) \quad (1)$$

where:

C is the channel capacity in bits per second
B is the channel bandwidth in Hertz
S is the average power of a signal received over bandwidth B
N is the average power of noise present in the channel
S/N is the SNR of the channel

After converting the log of base 2 to natural log and some manipulation, the following expression can be achieved

$${}^{C}/_{B} = 1.433\, {}^{S}/_{N} \quad (2)$$

In spread spectrum applications, the signal is below the noise floor. It can be safely assumed that SNR<<1. In that case

$${}^{C}/_{B} \approx {}^{S}/_{N} \quad (3)$$

Rearranging, we have

$${}^{N}/_{S} \approx {}^{B}/_{C} \quad (4)$$

This equation depicts that error-free transmission of information can be achieved in a given channel with fixed SNR by increasing the bandwidth of the transmitted signal. This relation is the basis of spread spectrum techniques [32].

### 2.4.2 Spread-Spectrum Basics

As shown above, by increasing the signal bandwidth, the detrimental effects of channel noise can be minimized. This technique of deliberately spreading the signal in frequency domain is called spread-spectrum technique. There are many different types of spread-spectrum techniques. One of the basic techniques is Direct Sequence Spread Spectrum or

DSSS, which uses a code of higher frequency than the actual data signal to spread the frequency spectrum of the transmitted signal [31].

This code, called the spreading code or chip sequence, is multiplied with the data signal before transmission as shown in Figure 10.

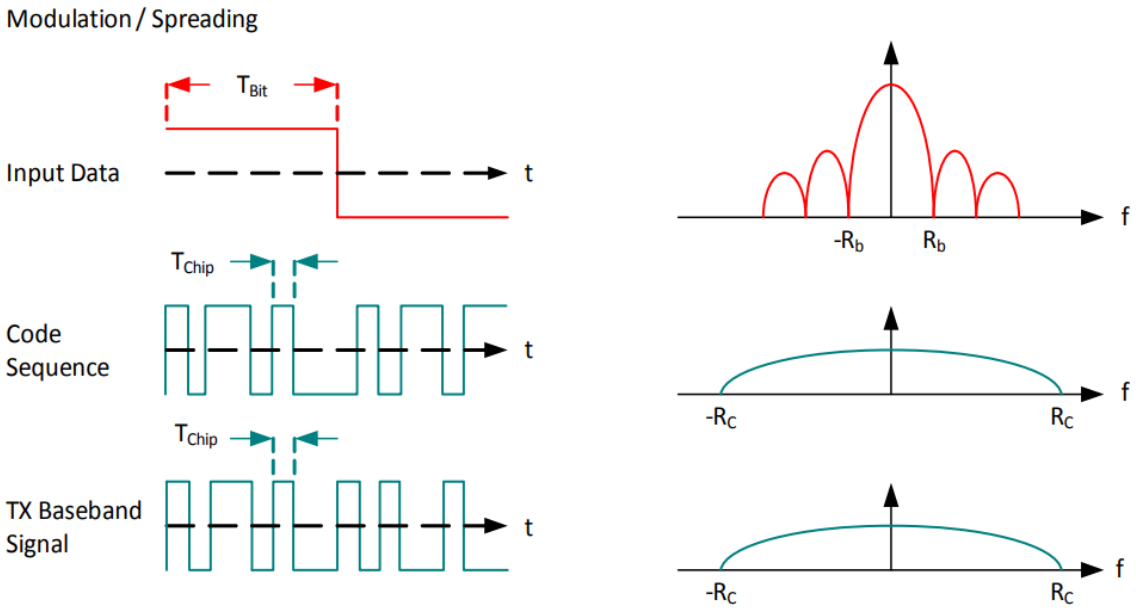


Figure 10. Spread Spectrum Modulation Using Coding Sequence.

The receiver obtains the original data signal by multiplying with the chip sequence again. In short, the transmitter spreads the signal by multiplying with the chip sequence and the receiver re-compresses the resulting transmitted signal by multiplying with the chip sequence again as shown in Figure 11.

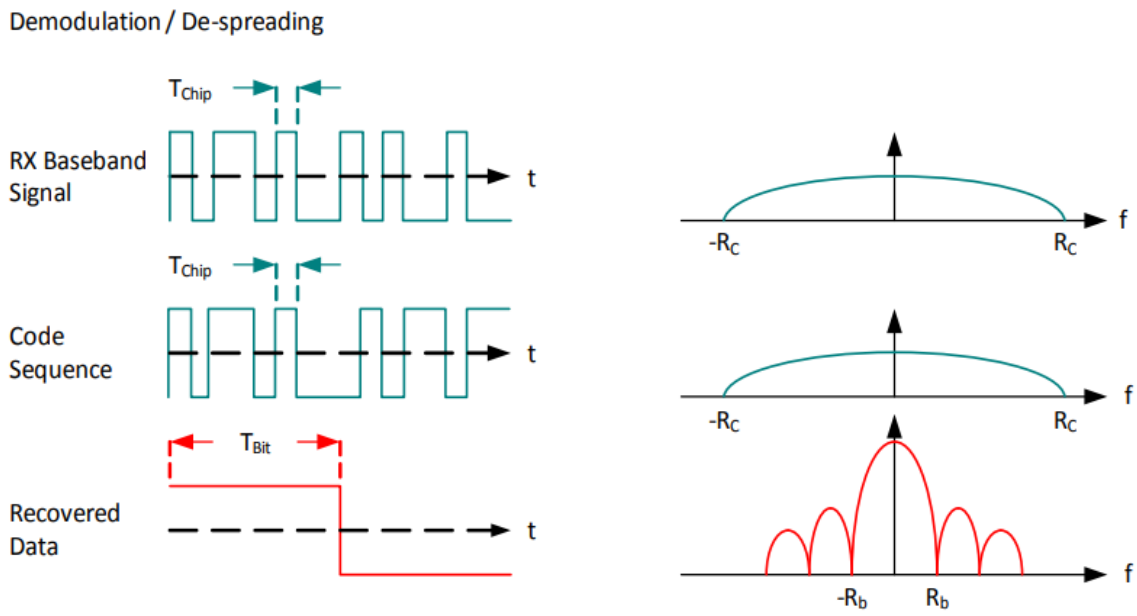


Figure 11. Spread Spectrum Demodulation Scheme.

The spreading factor of the chip signal depends on the ratio of the chip rate to that of the required data rate. This is called the processing gain ($G_p$), expressed as

$$G_p = 10\ log\ {}^{R_c}/_{R_b} \tag{5}$$

where:

$G_p$ = Processing gain in dB

$R_c$ = Chip rate in bits per second

$R_b$ = Required bit rate in bits per second

This process not only makes the signal more resistant to channel noise but also to interference from other signals. The processing gain allows for correct recovery of the original signal even when the SNR is negative. Any interference is spread out beyond the bandwidth of the data signal which makes it easier to filter it out.

One of the biggest challenged of implementing DSSS for low-cost and low power applications is the need of an accurate clock source. Moreover, longer chip codes require more time at the receiver for correlation over the whole length of the sequence. This not only makes receivers more complex but also increases the on time of the receivers. The receiver side has to stay always on to keep in sync with the transmitter which is not an option for low-power applications.

### 2.4.3 Chirp Spread-Spectrum (CSS)

A chirp is a sinusoidal signal of increasing or decreasing frequency. It relies on the linear nature of chirp signals for spreading the bandwidth spectrum of a transmitted signal. A chirp of increasing frequency is called an up-chirp as shown in Figure 12.

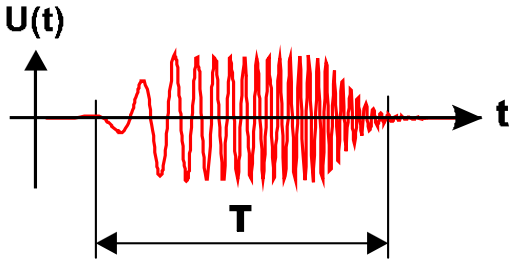


Figure 12. Up-Chirp in Time Domain.

A chirp of decreasing frequency is called down-chirp as shown in Figure 13.

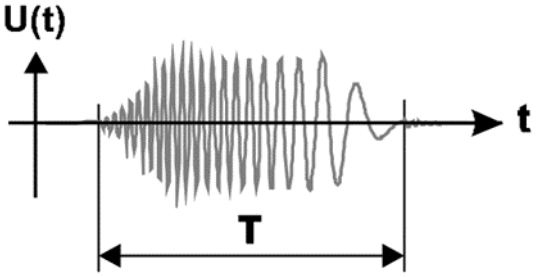


Figure 13. Down-Chirp in Time Domain.

Up-chirp has positive chirp value whereas down-chirp has negative chirp value. The change in frequency can either be linear or exponential. The bandwidth of a chirp signal is the difference between the starting frequency and the frequency at the end.

#### 1. Pulse Compression

Pulse compression is a process by which a long duration pulse with low peak power is converted into a short duration pulse with high peak. Chirps allow for a very straightforward pulse compression by correlation using a matched filter. The output of correlation with a matched filter is a pulse with combined power of the chirp pulse over its whole duration as we can see in **Error! Reference source not found.**. This results in a high processing gain and distance resolution [33].

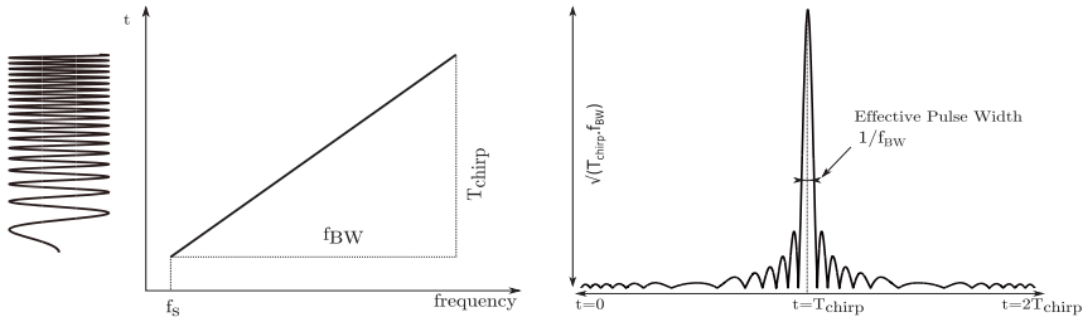


Figure 14. Chirp pulse and resulting pulse after compression [33].

Figure 15 shows the building blocks of CSS system. Data is modulated using up and down chirps at the transmitter and demodulated using correlation with matched filters for pulse compression and retrieved as high energy sharp pulses which can be easily decoded.

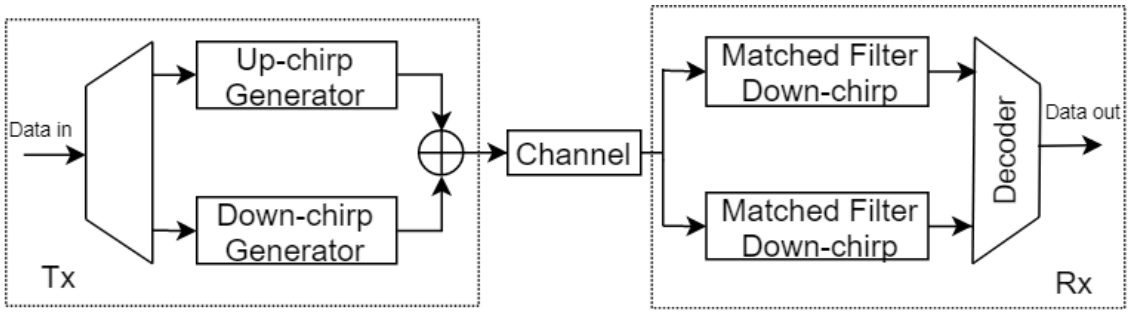


Figure 15. CSS Block Diagram [33].

One of the most important properties of CSS is range-data rate scalability. Chirp based spread spectrum can be used to spread the signal both in time and frequency independently.

### 2. Frequency Spreading

As explained in the previous section, spreading the signal bandwidth can help in reducing the detrimental effects of channel noise and make the signal more immune to interference. Using chirp pulses of much higher bandwidth, the bandwidth of signal can be scaled up as required.

### 3. Time Spreading

Since the bandwidth of chirp signals only depends on starting and ending frequency of the chirp, the data rate of chirp modulation can be scaled up and down independently of the bandwidth. Thus, it's possible to freely choose the bandwidth and data rate of the

resulting signal and adjust the BT product as required. This can result in very robust signals with high BT product.

Such flexibility allows for the design of highly scalable technology where range and data rate can vary rapidly according to the requirements of the system.

CSS systems can vary these according to the application requirements on the go. Other features include interference robustness, multipath resistance, low power, low latency owing to the fact that CSS doesn't need any synchronization. Due to these properties, CSS has been adopted for IEEE 802.15.4 standard for low rate wireless personal area networks [31].

### 2.4.4 LoRa Chirp Spread-Spectrum

The modulation scheme used in LoRa PHY is derived from traditional CSS, addressing the issues with DSSS systems. The signal in LoRa is modulated using a chirp signal of continuously varying frequency. The eliminates the need to have an accurate clock and always on receivers for synchronization. Both the frequency offset and the timing offset are equivalent for transmitter and receiver side. This allows for a much simpler PHY capable of operating with low-power requirements and robust communication link [31].

The bandwidth of the chirp signal generated for modulation is equivalent to that of the signal. The original data signal is first chipped using a signal of a higher rate. The resulting signal is then modulated using the chirp signal [31].

The LoRa modulation bit-rate can be defined as

$$R_b = SF \times {}^{1}/_{T_s} \tag{6}$$

where:

$R_b$ is the bit-rate in bits per second
SF is the spreading factor ranging from 7 to 12
$T_s$ is the symbol period in seconds

The symbol period can be defined as

$$T_s = \frac{2^{SF}}{BW} \tag{7}$$

where BW is the bandwidth of the modulated signal in Hertz,

As seen in the above relation, the bit-rate and symbol period is inversely proportional to each other, related by the spreading factor.

The chip rate of LoRa modulation can be defined as

$$R_c = R_s \times 2^{SF} \tag{8}$$

where

$R_c$ is the chip rate in chips per second
$R_s$ is the symbol rate in symbols per second

The symbol rate is the reciprocal of a symbol period

$$R_s = \frac{1}{T_s} = \frac{BW}{2^{SF}} \tag{9}$$

So

$$R_c = \frac{BW}{2^{SF}} \times 2^{SF} \tag{10}$$

So LoRa modulation sends “one chip per second per hertz”. Moreover, variable length error correction is also used in LoRa for increased robustness by trading off the data rate. So, the resultant achieved data rate of LoRa modulation scheme is given as:

$$R_b = SF \times \frac{\frac{4}{4+CR}}{\frac{2^{SF}}{BW}} \tag{11}$$

where

CR is the code rate for error correcting code ranging from 1 to 4
SF is the spreading factor of LoRa Modulation ranging from 7 to 12
BW is the bandwidth of LoRa Modulation in Hertz

### 2.4.5 Salient Features of LoRa Modulation

#### 1. High Bandwidth Time Product

LoRa modulation can achieve bandwidth-time product >1. This, when combined with asynchronous signalling, makes LoRa signals highly robust to both in a band and out of band interferences. LoRa modulation can achieve out of channel selectivity of up to 90dB and in-band rejection of up to 20dB [31].

#### 2. Bandwidth Scalability

LoRa modulation is capable of adapting to both narrowband and wideband applications owing to the inherent scalability of frequency and bandwidth. LoRa modules can easily be configured to suit any applications by changing the values in few configuration registers.

#### 3. Low Energy Consumption

LoRa modulation inherits the constant envelope modulation property of FSK modulation scheme hence it is possible to use the same low-cost and efficient power amplifier stages. Moreover, the processing gain of the chirp spread spectrum allows the output power of the transmitter to be reduced without deteriorating the link budget.

#### 4. Multipath Robustness

Owing to the broadband nature of chirp modulation scheme, LoRa is resilient to multipath and fading effects in urban environments.

#### 5. Long Range

LoRa signal can achieve much longer range as compared to other schemes like FSK on the same power. When combined with the aforementioned robustness properties, the link budget of LoRa can translate to four times enhancement in range.

#### 6. Doppler Effect Resistance

Since LoRa modulation is asynchronous in nature and there is no need for an accurate clock reference and synchronization, frequency shifts caused by doppler effects are easily mitigated. This makes LoRa ideal for mobile applications.

### 7. Network Capacity Enhancement

Multiple spread signals can be transmitted simultaneously over the channel since LoRa modulation uses orthogonal spreading factors. Signals of different spreading factors are filtered out as noise at the receiver end.

### 8. Localization Features

LoRa has the ability to differentiate between time and frequency errors which allows for various positioning and ranging applications.

## 2.4.6 LoRa Modules

LoRa PHY is a proprietary IP owned by Samtech, which also provides chip-based solutions for implementing LoRa transceivers. Moreover, Samtech licenses the IP to other companies for promoting the technology in the IoT paradigm. Currently, there are many radio chips available from Samtech for different application scenarios. The most basic series is the SX127x series of radio chips [34]. These are low-cost solutions intended for simple devices requiring minimal computation and processing. The SX130x series is intended for more advanced gateway solutions capable of decoding multiple signals on different frequencies at a time. The Figure 16 shows a block diagram of the SX127x series.

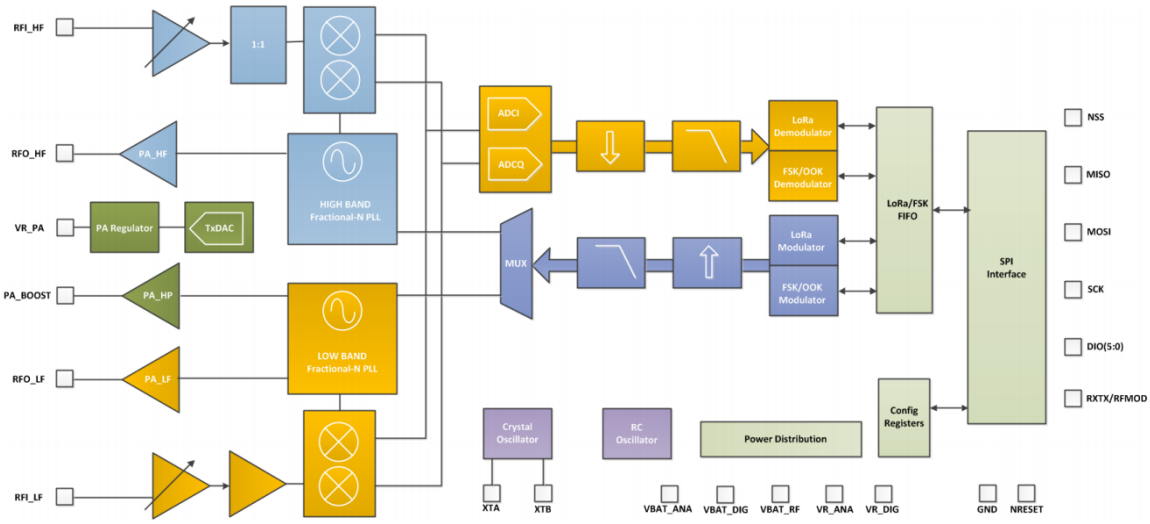


Figure 16. Samtech SX127x Series Block Diagram [34].

The SX127x series is capable of up to -148dBm sensitivity using only a low-cost crystal. Moreover, the SX127x series also includes a built-in 20dBm PA allowing for ultra-long-range communication links. The main interface is an SPI bus interface and 6

programmable digital I/O pins. The SX127x series chips can be configured through a series of config registers and one FIFO register for handling transmission and reception data streams.

On the RF side, there is a half-duplex transceiver operating at a low intermediate frequency. The received signal is first amplified by a low noise RF amplifier. Afterwards, the signal is converted to a differential mode for improved linearity and harmonic rejection before down-conversion. Two delta-sigma analog to digital converters convert the data for digital signal processing.

### 2.4.7 LoRa PHY Packet

The packet structure used by LoRa modems is shown in Figure 17.

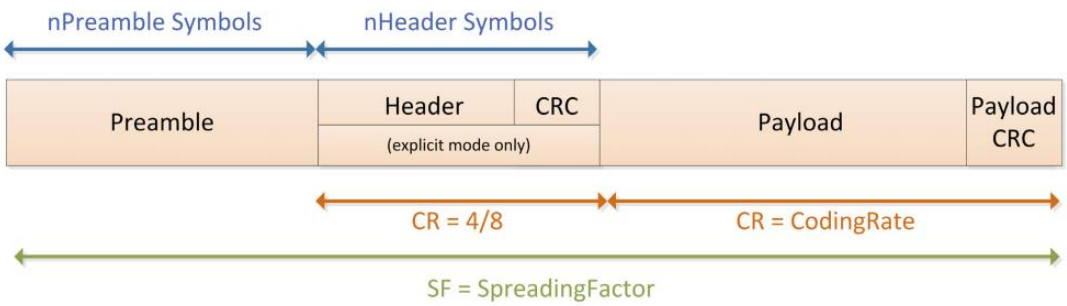


Figure 17. LoRa PHY Packet Structure [35] .

There are three main parts of a LoRa packet

- Header
- Payload
- Preamble

**1. LoRa Packet Header**

There are two main types of packets:

- Explicit Header Packets
- Implicit Header Packets

The difference between implicit and explicit header packets, as the name suggests, is in the header. The explicit packets include a short header which contains information about the size of the packet, coding rate used, and CRC settings.

#### i. Explicit Header

The header provides information about the payload size, error correction code used, and CRC mode. This is the default header type in LoRa PHY. Explicit headers are transmitted using the maximum error correction code rate of 4/8 along with its own CRC for robust identification of invalid headers on a receiver side.

#### ii. Implicit Header

In this mode, LoRa packets are sent without any header information. This mode is suitable for applications where the size of payload, error correction code and CRC code settings are fixed. This mode helps reduce transmission time but packet settings need to be configured beforehand on both transmitter and receiver sides.

### 2. LoRa Packet Payload

The payload field of a LoRa packet has a variable length. It contains the actual data transmitted and CRC, if operating in CRC mode. The forward error connection coding rate of the payload field depends upon the coding rate chosen.

### 3. LoRa Packet Preamble

Every LoRa packet starts with a preamble field. By default, the preamble is a 12-symbol long sequence used to synchronize the receiver. The length of the preamble, however, can be modified according to the application requirements but it has to be same on both transmitter and receiver side. LoRa modules support a preamble length of 10-65540 symbols.

## 2.4.8 LoRa Transmission Time

Time on an air of LoRa packet depends on three main factors.

- Error Correction Coding Rate or CR
- Spreading Factor or SF
- Bandwidth or BW

For a given combination of these three variables, the resulting transmission time for LoRa packets can be calculated using the following procedure. As defined earlier, symbol rate is given as

$$T_s = \frac{1}{R_s} \tag{12}$$

The total duration of transmission of a single LoRa packet is consists of time spent transmitting the preamble and time required to send the actual packet. The time duration for preamble transmission is as follows

$$T_{preamble} = (n_{preamble} + 4.25)T_s \tag{13}$$

where:

$n_{preamble}$ is the length of preamble ranging from 10 to 65540

$T_{preamble}$ is the preamble transmission time

$T_s$ is the symbol rate of a preamble

Similarly, the transmission time of payload also depends upon the number of symbols in the actual payload. The symbol length of payload with an explicit header is given as

$$n_{payload} = 8 + \max\left(ceil\left[\frac{(2P+4CRC-SF+7)}{(SF-2D)}\right](CR+4), 0\right) \tag{14}$$

And for implicit header

$$n_{payload} = 8 + \max\left(ceil\left[\frac{(2P+4CRC-SF+2)}{(SF-2D)}\right](CR+4), 0\right) \tag{15}$$

Where:

SF is the spreading factor selected

D is 1 when data rate optimization is enabled, zero otherwise

CR is the error correcting coding rate selected

CRC is one if CRC is enabled

P is the number of payload bytes (1-255)

The resultant transmission duration for the payload is given by

$$T_{payload} = n_{payload} \times T_s \tag{16}$$

The total transmission time is given as

$$T_{packet} = T_{payload} + T_{preamble} \tag{17}$$

## 2.5 LoRa Networking Protocol

Since LoRa is a physical layer protocol, the choice of upper layers of the OSI model is very flexible as shown in Figure 18.

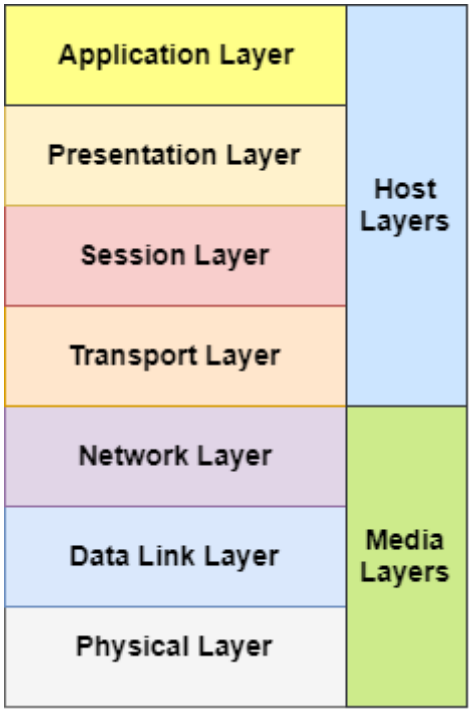


Figure 18. OSI Model.

For low power monitoring and WSN applications only deal with the media layers of the OSI model. Higher levels are not relevant for low power wide area networks since most sophisticated networking problem in such applications is the efficient routing of traffic from endpoints to the sink. For LoRa applications, the most popular protocol is the LoRaWAN protocol.

### 2.5.1 LoRaWAN

LoRaWAN is a protocol specification by the Lora Alliance for a “star of stars” type WSN with low power battery operated endpoints collecting data. LoRaWAN networks consist of gateways that act as relay nodes between endpoints and cloud. LoRaWAN protocol allows secure communication framework relying on cryptographic keys. LoRaWAN endpoints are allowed to transmit data at any time, using any data rate and any channel available as long as certain rules are followed [36].

LoRaWAN devices are supposed to follow a certain standard for available functionality. Devices are categorized into three classes according to the supported functionality as shown in Figure 19.

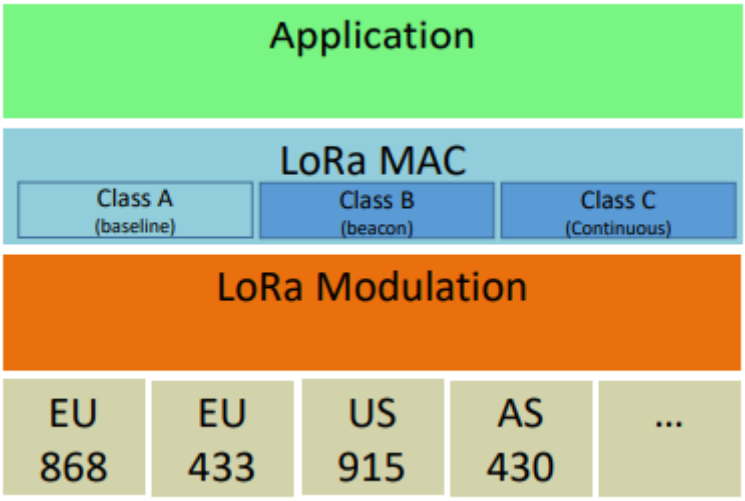


Figure 19. The LoRaWAN Stack [36].

The most basic LoRaWAN class, called Class A, specifies a bi-directional endpoint which allows for two short receiving windows after every transmission. Class A devices require the least amount of power for operation. Class B devices have scheduled reception slots for receiving data from the server. In addition to Class A type reception capability, Class B devices allow for extra reception slots using a time-synchronized beacon from the gateway. Class C devices are continuously receptive to any messages from the gateway, hence, they consume the most power.

Although LoRaWAN is a sophisticated and standardized protocol for implementing LoRa based sensor networks, an overkill for simple P2P type networks. The smart agriculture system implemented in this work needs a simpler and more efficient protocol for achieving its goal. For this reason, a variation of LoRa protocol by Libelium was utilized. This results in cheaper modules with more control and flexibility over the design of the sensor network.

### 2.5.2 Libelium LoRa Protocol

This protocol only contains the datalink layer of the OSI model. The topology used is single-hop star topology with one gateway connected to several endpoints. A network created with this protocol has one central node and multiple sensor nodes. Every network

operates on a set of prespecified parameters and an endpoint connected to the network must have matching parameters. These important parameters are.

### 1. Node Address

Each node has a unique address associated with it. This address is 8-bit long, allowing for up to 255 devices. The nodes have to be assigned addresses manually. Address 0 is dedicated to broadcast and address 1 is dedicated to the central gateway. Endpoints can have addresses ranging from 2 to 255.

### 2. LoRa Frequency

This protocol supports two frequency bands from ISM bands i.e. 868MHz and 900MHz. The choice of a frequency band used depends on the country of application. Each band is further divided into multiple channels.

### 3. LoRa Channels

The 868MHz band is divided into 8 channels as shown in Figure 20 whereas the 900MHz band is divided into 13 individual channels for transmission and reception of LoRa signals as shown in Figure 21.

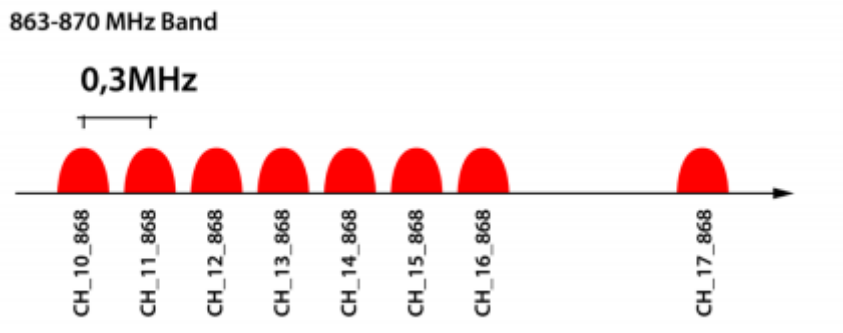


Figure 20. LoRa Channels, 868-870 MHz [37] .

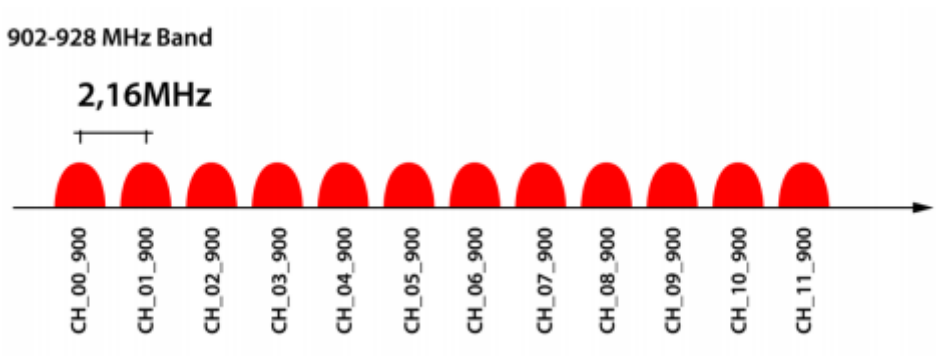


Figure 21. LoRa Channels, 902-982 MHz [37].

## 4. Network Mode

This protocol defines a set of Network Modes using various combinations of coding rate, bandwidth and spreading factor. These modes vary from maximum range to maximum throughput. Depending on the application, a suitable mode can be chosen from a set of 10 modes given below in Figure 22.

| Mode | BW | CR | SF | Sensitivity (dB) | Transmission time (ms) for a 100-byte packet sent | Transmission time (ms) for a 100-byte packet sent and ACK received | Comments |
|---|---|---|---|---|---|---|---|
| 1 | 125 | 4/5 | 12 | -134 | 4245 | 5781 | max range, slow data rate |
| 2 | 250 | 4/5 | 12 | -131 | 2193 | 3287 | - |
| 3 | 125 | 4/5 | 10 | -129 | 1208 | 2120 | - |
| 4 | 500 | 4/5 | 12 | -128 | 1167 | 2040 | - |
| 5 | 250 | 4/5 | 10 | -126 | 674 | 1457 | - |
| 6 | 500 | 4/5 | 11 | -125,5 | 715 | 1499 | - |
| 7 | 250 | 4/5 | 9 | -123 | 428 | 1145 | - |
| 8 | 500 | 4/5 | 9 | -120 | 284 | 970 | - |
| 9 | 500 | 4/5 | 8 | -117 | 220 | 890 | - |
| 10 | 500 | 4/5 | 7 | -114 | 186 | 848 | min range, fast data rate, minimum battery impact |

Figure 22. LoRa Network Modes [37].

## 5. Packet Structure

The packet structure of this protocol is shown in Figure 23.

| Destination Address | Packet Type | Source Address | Sequence Number | Payload |
|---|---|---|---|---|
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | variable |

Figure 23. LoRa Protocol Packet Structure.

The destination address field contains the 8bit address of the destination node. The source address is also an 8bit field containing the address of the node sending the packet. A

sequence number is an 8bit field containing the packet number. The packet type field contains information about the type of data in the payload and some flags as shown in Figure 24.


Figure 24. Packet Type Field.

The first four bits of the packet type field contains a code for identifying the packet. This code is 1 for the DATA packet and 2 for ACK packet. If the source requires an ACK from the destination, the ACK flag is set. The encryption flag is set if the data in the payload is encrypted. If the gateway requires a key for authorization, the AppKey flag is set indicating the payload contains the key. A Binary flag is set to indicate binary data.

### 2.5.3 LoRa Limitations

LoRa is designed specifically for applications that require transmission of small amounts of data periodically using a minimal amount of power. The three key requirements that LoRa tries to address are battery life, simplicity, and range. It is not suitable for applications that require low latency and high data rate [38].

Another disadvantage of LoRa lies in the use of unlicensed radio bands. Although the use of ISM bands is advantageous to some extent, for practical applications, a LoRa network can be disrupted if other users in the same area start using the same frequency band in big volume. There is no way to curtail the usage of the same frequency band hence there is no control.

# 3 Chapter 3

This section describes the hardware and software resources utilized to build the smart agriculture monitoring system. The idea behind the design process was to utilize low cost, commercially available solutions to reach the goal.

## 3.1 Hardware: LoRa Endpoint Design

Each LoRa endpoint has three important parts as shown in Figure 25.

- Microcontroller platform with debugging support
- Sensor platform for temperature, humidity and pressure
- LoRa Transceiver

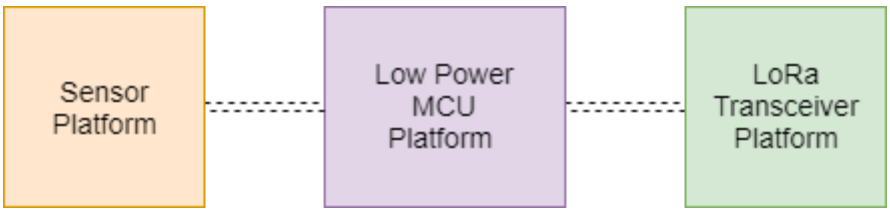


Figure 25. LoRa Endpoint.

### 3.1.1 MCU Platform

There are dozens of microcontroller families available in the market. The key factors affecting the choice of MCU platform are as follows.

### 3.1.2 Performance

Embedded microcontroller families support a variety of clock speeds and performance levels. A faster clock roughly means more work done in a certain amount of time but high clock speeds result in more power required to run the processor.

Microcontrollers can also be classified according to the size of registers. There 4bit, 8bit, 16bit, 32bit and even 64bit microcontroller families available. Generally, more bits mean more data can be processed in a given time.

Another important feature affecting the performance of microcontroller families in the instruction set architecture. For example, a microcontroller with a dedicated floating-point calculation unit can provide native support for floating point number in its instruction set. This results in better, more efficient performance as doing such calculations in software requires more instructions and bigger memory footprint. Some notable architectures are ARM, AVR, PIC, 8051, and MSP430. For a LoRa endpoint, the most important feature of microcontroller architecture is the support for low power modes. An endpoint collects and transmits data in a periodic fashion and to save power, the device should consume a minimum amount of power in between consecutive transmissions.

### 3.1.3 Development Environment

The support for an integrated development environment with a robust compiler and debugging support greatly affects the development time and performance. Any architecture is as good as the compiler it comes with.

### 3.1.4 Memory

The amount of memory included in a microcontroller greatly affects the amount of software that can be written. For LoRa, the flash memory of the controller should be enough to support both the protocol and drivers for sensor platform.

### 3.1.5 Peripherals

Peripherals let a microcontroller interact with the outside world. LoRa modules require a dedicated SPI master at the microcontroller for interfacing and configuration. The choice of sensor platform also dictates the peripherals required. Examples of peripherals are UART, GPIO, I2C, SDIO etc.

### 3.1.6 Cost

Cost is another important factor in considering the choice of microcontroller. Cost doesn't only include the price of individual chips but also the cost of development platform, IDE licensing etc.

### 3.1.7 Debugging

In system programming and debugging support is another important feature to consider when choosing microcontroller families. Debuggers greatly reduce development time by allowing the designer to quickly find and solve any problems in the firmware.

Considering all the factors explained above, MSP430 family by Texas instruments was chosen for the design of LoRa endpoints. MSP430 family consists of 16bit microcontrollers designed specifically for low power applications. The launchpad ecosystem by TI allows for a quick and hassle-free solution. Low-cost development boards within system debuggers are easily available in the market.

## 3.2 MSP430F5529LP Launchpad

MSP40F5529LP Launchpad shown in Figure 26 belongs to MSP430 family.

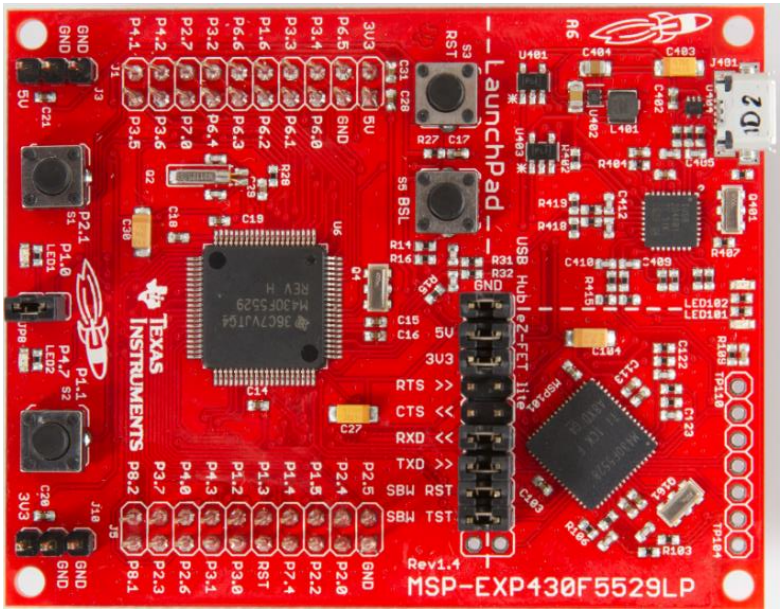


Figure 26. MSP430F5529 Launchpad.

Salient Features:

- 16bit MSP430 Family
- Up to 25MHz clock
- Ultra-low Power Mode
- 128Kb Flash
- 8Kb RAM
- 4 Universal Serial Interfaces (UART, SPI, I2C)
- 1.8V to 3.6V Operating Range
- TI ez-FET Debugger
- Isolation jumpers for accurate power measurement

The MSP43F5529LP platform is a great combination of performance and efficiency. With more than adequate memory, 25MHz clock, license free IDE, and Launchpad ecosystem, and built in debugger, this development platform is quite suitable for the development of LoRa endpoints [39].

## 3.3 Sensor Platform

For the choice of the environmental sensor platform, the key features to be considered are performance, cost and interface. BME280 by Bosch is an integrated environmental sensor platform developed specifically for mobile applications as shown in Figure 27. It combines temperature, humidity and pressure sensor in one chip, specifically designed for low current consumption. There are two digital bus interfaces available: SPI and I2C [40].

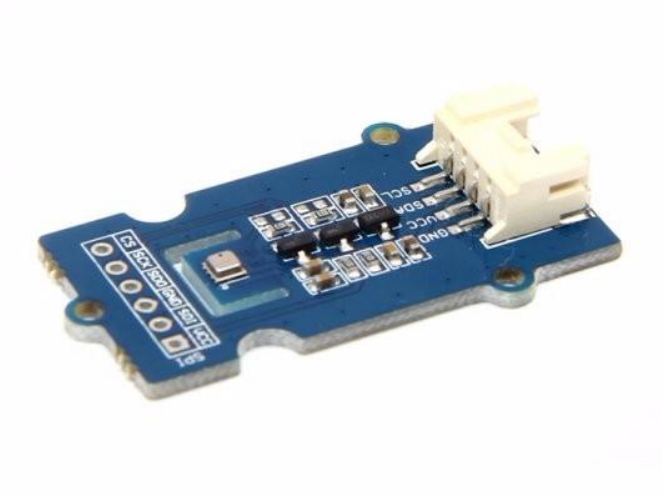


Figure 27. Bosch BME280 Sensor Platform.

## 3.4 Lora Platform

Draguino is a LoRa platform which uses HopeRF's LoRa chipset to achieve a sensitivity of -148dBm using a low-cost platform as shown in Figure 28. The HopRF chips are completely compatible with Samtech's SX127x series of chipsets. The Draguino board also features a 14 dBm highly efficient power amplifier stage and built-in bit synchronizer for clock recovery [41].

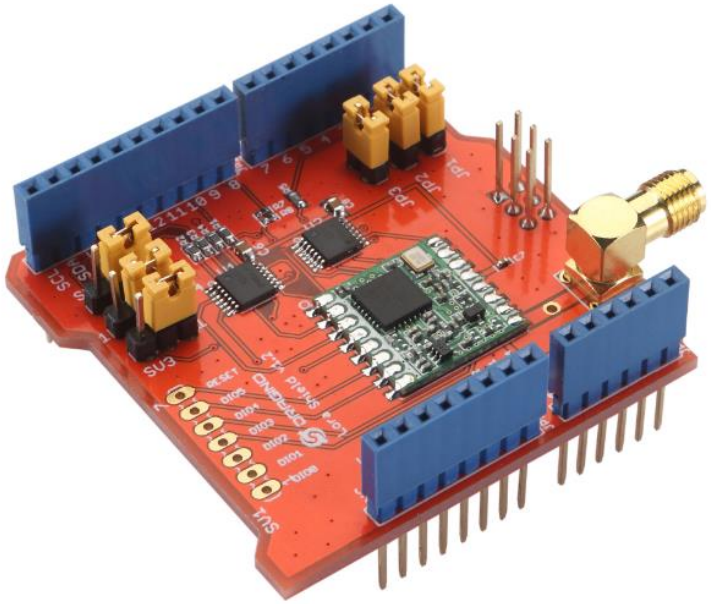


Figure 28. Draguino V1.4.

## 3.5 Hardware: LoRa Gateway Design

The LoRa gateway is a bridge between all the endpoints and the web interface as shown in Figure 29. It has to be powerful enough to collect LoRa traffic from all endpoints and Host a web interface through Wi-Fi connectivity.

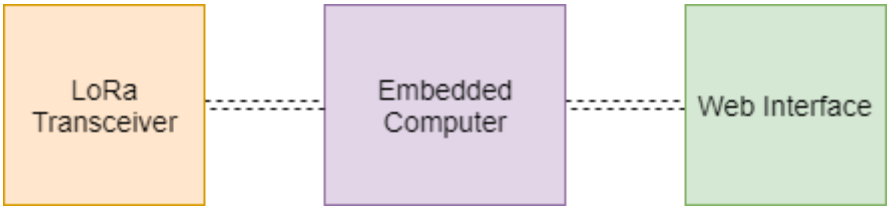


Figure 29. LoRa Gateway.

A Raspberry Pi running embedded Linux(Raspbian) was chosen as the main platform for a gateway. Raspberry Pi is an inexpensive single board computer with a rich ecosystem as shown in Figure 30.

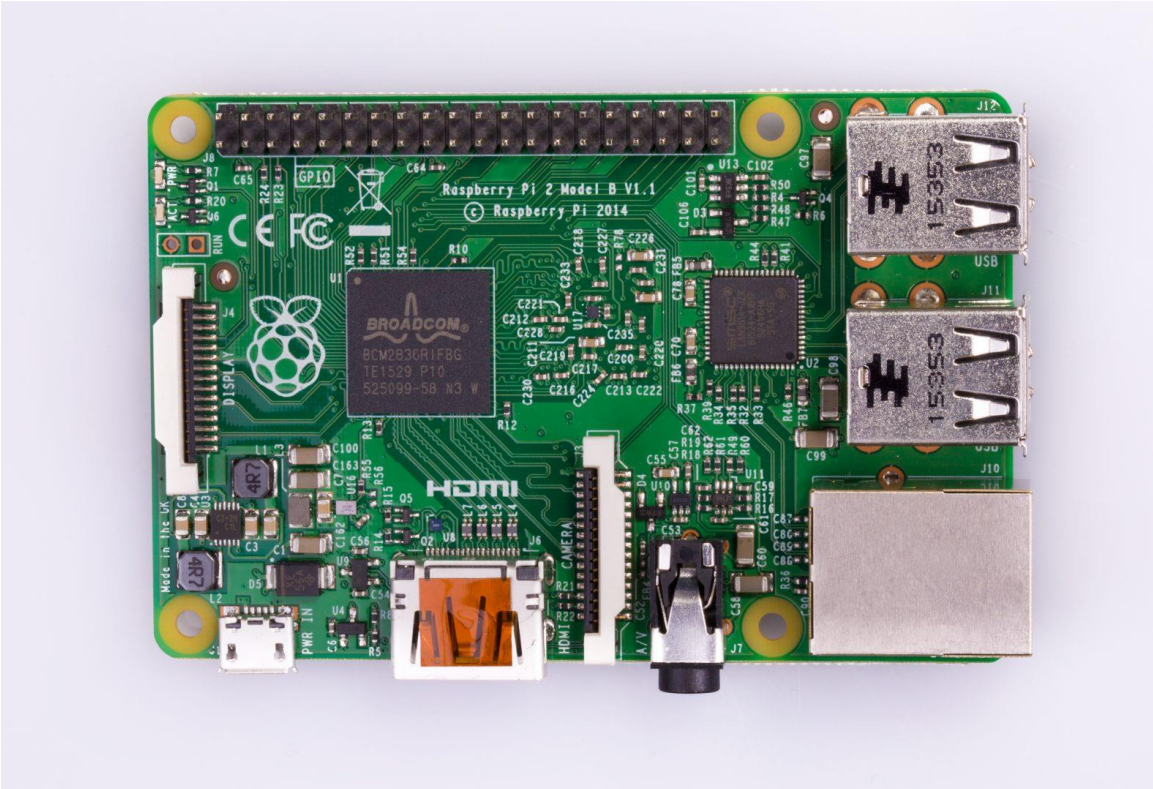


Figure 30. Raspberry Pi Model 2B.

Similar to the endpoint design, a HopeRF based LoRa platform specifically designed for Raspberry Pi was used. The Draguino LoRa Hat for Raspberry Pi also features a built-in GPS for automatic positioning of Lora Gateway. The Following Figure 31 shows the Draguino LoRa Hat mounted on a Raspberry Pi 2 [42]-[43].

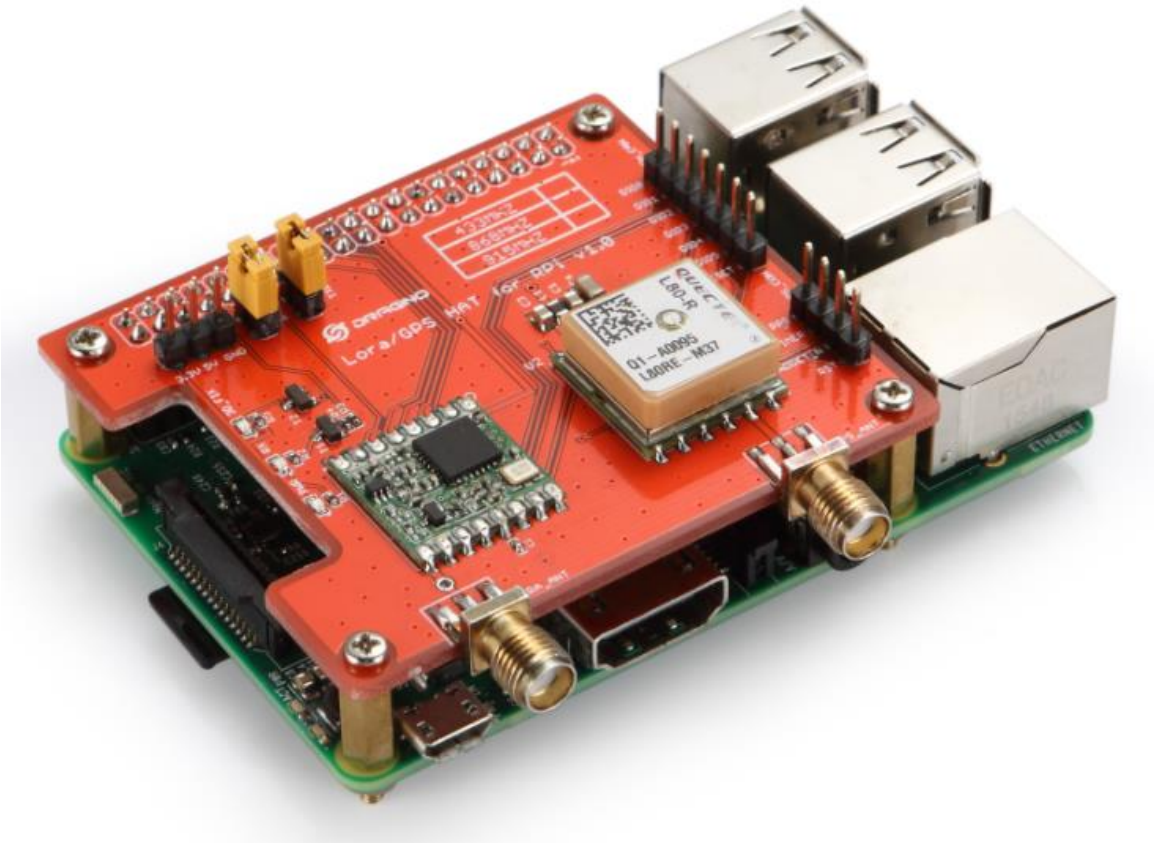


Figure 31. Raspberry Pi with LoRa Module Attached.

For web connectivity, an inexpensive Wi-Fi USB module was used.

## 3.6 Software Resources

### 3.6.1 Code Composure Studio

The main Integrated Development Environment used for developing the endpoint was Code Composure Studio by Texas Instruments. Figure 32 represents Code Composer Studio logo. It is primarily based on Eclipse, supports the MSP430 platform and allows for JTAG debugging [44].


Figure 32. Texas Instruments Code Composure Studio [44].

### 3.6.2 Raspbian Linux Distribution

Raspbian is an open source Linux distribution for Raspberry Pi, developed by the Raspberry Pi foundation. It's a Debian based distribution optimized specifically for Raspberry pi single board computers. Raspbian is the main operating system of the LoRa Gateway.

### 3.6.3 WAZIUP IoT Framework

WAZIUP is an open source research project for developing low-cost IoT solutions for sub-Saharan Africa. WAZIUP IoT framework includes a low-cost implementation of LoRa gateway using Libelium like protocol. The open source libraries for raspberry pi gateway were used to implement the LoRa gateway.

### 3.6.4 Tera Term

An open source terminal emulator program with SSH and serial support. Tera Term was not only used to obtain serial output from the endpoint during a design process but also for configuring the gateway remotely through SSH.

# 4 Chapter 4

## 4.1 HopeRF LoRa Library

A C Library for HopeRF Lora Modules was created consisting of two files “LORA.c” and “LORA.h”. The detail of some important functions is given below. Moreover, the complete code is available in the Appendix Section.

**void LORA_Init**()

This function initializes the LoRa module using the SPI interface as shown in Figure 33. It begins by cycling the module reset followed by verification of module version. Valid HopeRF modules have a value of 0x12 stored in the version register. After successful verification, receiver chain of LoRa module is calibrated at both LF and HF band. Afterwards, the maximum current of LoRa module is set to 100mA by writing 0x1B in the overcurrent protection control register (RegOcp). Since the HopeRF modules support both FSK and LoRa modulation, we need to put the modules in LoRa mode by initializing the operation mode control register. After successfully initializing LoRa mode, CRC is enabled by modifying the Modem Configuration Registers. The sync word for LoRa protocol is set to 0x12.

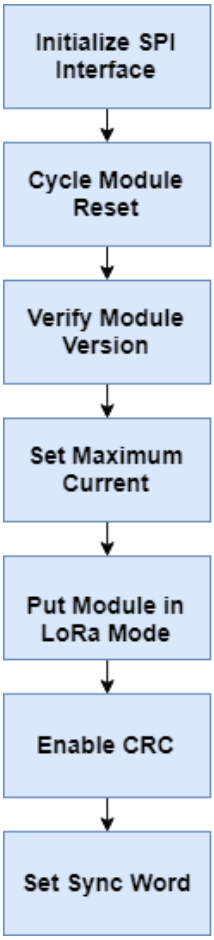


Figure 33. LoRa Module Initialization Routine.

**void LORA_Set_Mode(uint8_t libelium_mode)**

This function sets the appropriate bandwidth, spreading factor and coding rate of LoRa modulation according to the predefined modes explained in the previous chapter. For this project, the testing was done in Mode 1, which corresponds to a bandwidth of 125kHz, a coding rate of 4/5, and spreading factor of 12.

**LORA_Set_Frequency(uint32_t frequency)**

This function sets the frequency channel of the module by writing the corresponding binary values to the frequency registers of the module. There are three frequency registers corresponding to three bytes of frequency value, namely, RegFrfMsb, RegFrfMid, and RegFrfLsb. For this project, channel 10 of 868MHz frequency was used corresponding to a value of 0xD84CCC.

**Void LORA_Set_Power_Db(int8_t dbm)**

This function sets the output power of the module by writing appropriate value to power amplifier config register. The input power in dBm is converted to the corresponding value

for configuration registers and written to the registers through the SPI interface. A value of 14dBm was used for this project.

**void LORA_Set_Node_Address(uint8_t addr)**

This function sets the address of the node. Node address of the LoRa Module can be set by writing the 8bit address to RegNodeAddr. Moreover, this function also sets the broadcast address to a default value of 1 by modifying the broadcast address register.

**void LORA_Set_Packet_Type(uint8_t type)**

This function sets the packet type of the packet being sent out by the endpoint. The library includes a global structure called "packet_sent" that contains all the information of the packet being sent out. This function initializes the corresponding "type" field of the "packet_sent" struct to the value passed to the function. For this project, DATA type packets were used with a type value of 0x10.

**void LORA_Send_Packet(uint8_t dest, char *payload,uint8_t length)**

This function takes the destination address, pointer to the payload array and length of data to be transmitted and sends a packet over LoRa. After initializing the sent packet structure, the transmission sequence of LoRa modem is performed as shown in Figure 34.

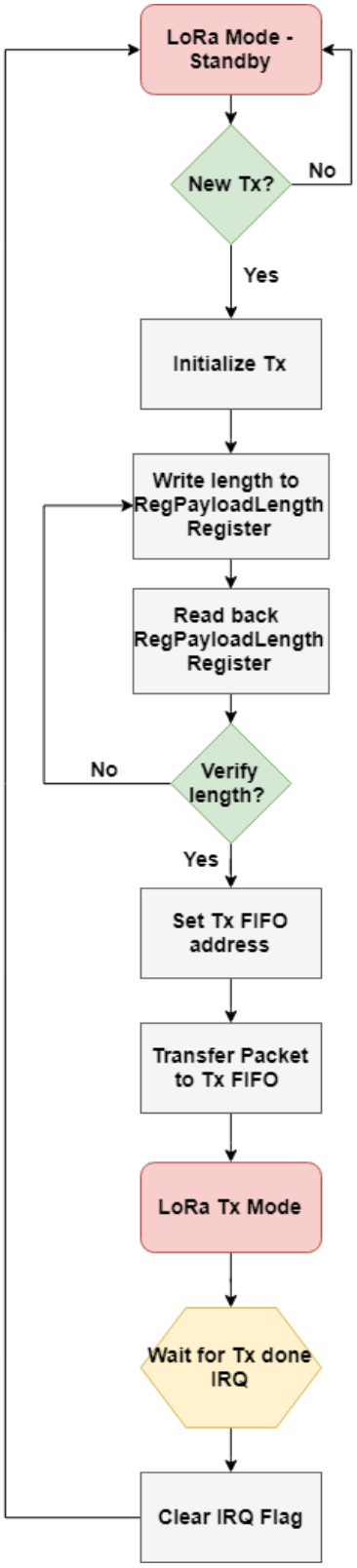


Figure 34. Transmission Sequence.

**void LORA_Receive_Packet(char *payload)**

This function pointer to the received payload array and receives over LoRa. The reception sequence is shown in Figure 35. The function is blocking and waits for the received data

IRQ flag. When the flag is set, it verifies that there was no CRC error in the received data and copies the data in the payload array.

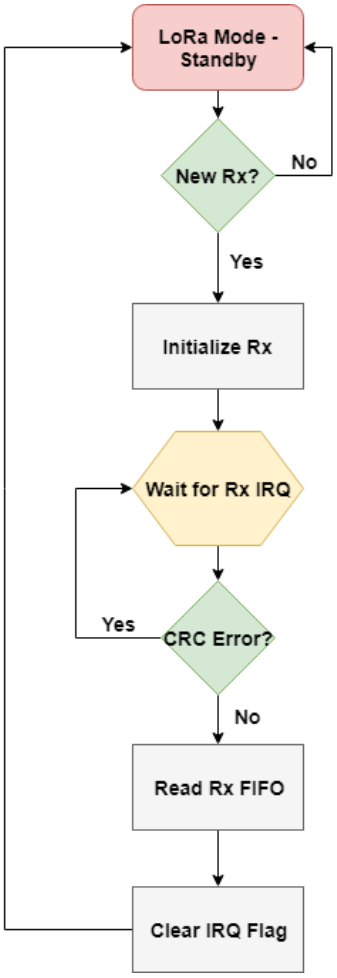


Figure 35. LoRa Reception Sequence.

The same library is used both for endpoint and gateway. The gateway is always in reception mode after proper initialization whereas the endpoint transmits data periodically and goes into standby mode in between transmissions. Notice that there is no ACK packets in this protocol since sending ACKs will result in endpoint busy waiting in reception mode. This is an inefficient process that results in a lot of wasted power on the endpoint side.

## 4.2 BME280 Sensor Library

BME280 sensor has an I2C interface and various configuration registers for configuring and acquiring the sensor data. Following are some important functions of the C library developed for this project.

**void BME280_Init**()

This function initialized the sensor through an I2C interface. First, the sensor is verified by reading the chip ID from the chip ID register. BME280 chip ID sensor contains 0x60 as default chip ID. If the ID is matched, the sensor is issued a soft reset by writing 0xB6 to the corresponding register. After the reset, the sensor calibrates itself. This process takes some time and sensor state can be checked through the status register. If the first bit of status register is set, the sensor is busy calibrating itself and cannot take further commands. After this process is complete and the busy status bit is zero, the factory calibration values of the sensor are read and stored in a factory_cal struct as shown in Figure 36. These values are to be used later for obtaining temperature, humidity and pressure information from raw sensor data. After reading the calibration values, sensor configuration registers are set to the default values. After calling this function, the sensor is ready and the following functions can be used to obtain sensor data in floating point precision.

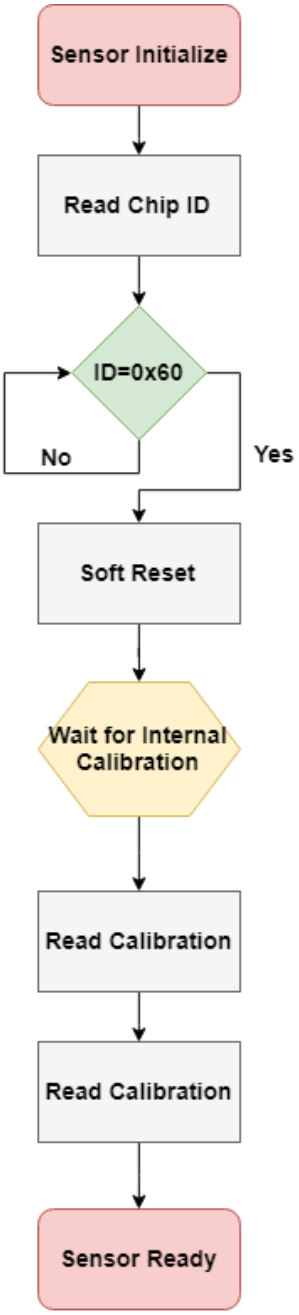


Figure 36. BME280 Sensor Initialization.

**float BME280_Read_Temperature(void)**

This function obtains the raw sensor data from the temperature ADC registers of BME280 and uses the factory calibration values to calculate the temperature in degrees Celsius with floating point precision.

**float BME280_Read_Pressure(void)**

This function obtains the raw sensor data from the pressure ADC registers of BME280 and uses the factory calibration values to calculate pressure in the atmosphere with floating point precision.

**float BME280_Read_Humidity()**

This function obtains the raw sensor data from the humidity ADC registers of BME280 and uses the factory calibration values to calculate relative humidity with floating point precision.

## 4.3 Web Interface

Using python, the data coming from the sensor is taken from the terminal of raspberry pi and saved into a text file. A simple web page takes the data from the text file and displays it on the browser. An apache server is setup on the Raspberry Pi which is connected to the network using WiFi. Using the local IP address of raspberry pi, this page can be accessed anywhere from the same network.

## 4.4 Gateway State Machine

The gateway is always in listening mode for any data coming on the network as shown in Figure 37.

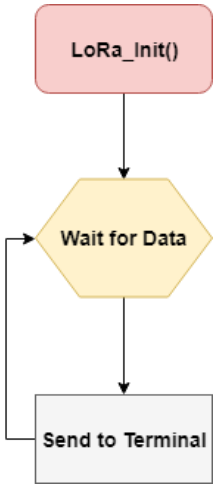


Figure 37. Gateway State Machine.

## 4.5 Endpoint State Machine

The endpoint initializes both the sensor and the LoRa module as we can see in Figure 38. Afterward it periodically takes sensor values and transmits them to the gateways after a delay.

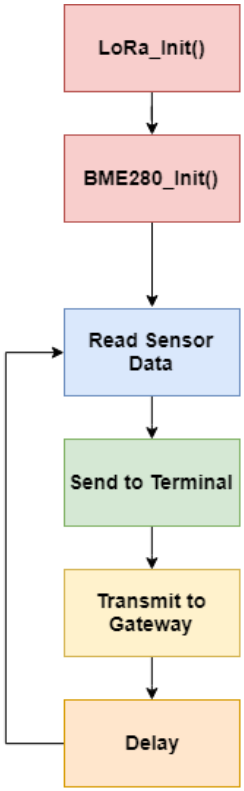


Figure 38. Endpoint State Machine.

# 5 Chapter 5

## 5.1 Results and discussion

The test setup includes one endpoint placed about 40meters away from the gateway. The endpoint sends sensor data to the gateway every minute. The length of the message sent by the endpoint is 67 bytes. Including the header, the total length of the message becomes 71 bytes. The LoRa mode used by both gateway and endpoint to setup the network was mode 1 i.e. 125kHz bandwidth, spreading factor of 12, and coding rate of 4/5. With an SNR of 6 and RSSI of -39, zero packet loss was observed in the setup.


Figure 39. Web Interface.

This setup is more of proof of concept of the design process involved in setting up a basic LoRa network. The protocol setup is useful for low-density low-cost applications with a focus on simplicity. There is a possibility of using LoRaWAN protocol directly without having to write the whole protocol from scratch but LoRaWAN networks need an expensive multichannel gateway capable of receiving data on various channels simultaneously. This not only makes the whole setup more complicated but also adds to the cost of setting up the network.

For future works, some suggested improvements for this setup include setting up carrier sense routines in the LoRa library, integration with the cloud using a suitable cloud API and the development of single PCB endpoints suitable for outdoor applications.

The LoRa modules used in this work have a built-in channel activity detection capability. The module detects the channel for any activity during this mode. The use of RSSI is not feasible for channel detection since LoRa signal is usually below the noise floor. The time

taken by the modem for channel activity detection depends upon the modulation settings used. This feature is important when the network is deployed in an area with other devices working in the same band. This is one of the drawbacks of using license-free ISM bands for deploying wireless sensor networks. Since this project focuses on the agricultural application in the rural areas, the chances of having other devices saturating the network are pretty low. Hence, for the current system, this functionality was omitted. Moreover, carrier activity detection keeps the end devices on for longer periods of time which reduces battery life.

## 5.2 Conclusion

Wireless sensor networks have a huge potential for smart monitoring systems. The ability to develop low cost and simple end devices for quickly deploying in the field makes WSNs very attractive for smart agricultural systems. Although there are various options for the choice of communication protocol and radio technology suitable for deploying wireless sensor networks, LoRa comes out on top as a robust, reliable, simplistic, low cost and highly flexible solution.

Most of the research in LoRa based sensor networks focuses on theoretical aspects of communication protocols and network efficiency. There is a gap in the literature about the engineering design and application of such systems for a real-world application. Moreover, most of the engineering work focuses on LoRaWAN as the networking protocol of choice for deploying LoRa based sensor networks. Although LoRaWAN is a proven and elaborate networking protocol for LPWAN applications, low-density networks can be deployed using extremely simple and effective protocols designed specifically for the application.

This thesis focused on design, development, and application of LoRa technology using low-cost solutions available in the market. A simple protocol for low-density agricultural monitoring system was design and implemented. The data from endpoints was channelled to a simple web interface through a Linux based single board computer.

# References


[1] Y. Song, J. Lin, M. Tang, and S. Dong, "An Internet of Energy Things Based on Wireless LPWAN," *Engineering*, vol. 3, pp. 460–466, 2017.

[2] "Top Trends in the Gartner Hype Cycle for Emerging Technologies, 2017 - Smarter With Gartner." [Online]. Available: https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/.

[3] J. P. Bardyn, T. Melly, O. Seller, and N. Sornin, "IoT: The era of LPWAN is starting now," in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, 2016, pp. 25–30.

[4] NGMN Alliance, "NGMN 5G White Paper," *Ngmn*, pp. 1–125, 2015.

[5] "Emerging-Technology-Hype-Cycle-for-2017_Infographic_R6A.jpg (1500×1269)." [Online]. Available: https://blogs.gartner.com/smarterwithgartner/files/2017/08/Emerging-Technology-Hype-Cycle-for-2017_Infographic_R6A.jpg.

[6] A. Bröring *et al.*, "New generation sensor web enablement," *Sensors*, vol. 11, no. 3, pp. 2652–2699, 2011.

[7] F. Nack, "An overview on wireless sensor networks technology and evolution.," *Sensors (Basel).*, vol. 9, no. 9, pp. 6869–96, 2009.

[8] M. F. Othman and K. Shazali, "Wireless Sensor Network Applications: A Study in Environment Monitoring System," *Procedia Eng.*, vol. 41, pp. 1204–1210, 2012.

[9] R. Sharan Sinha, Y. Wei, and S.-H. Hwang, "ScienceDirect A survey on LPWA technology: LoRa and NB-IoT," *ICT Express*, vol. 3, pp. 14–21, 2017.

[10] International Electrotechnical Commission *et al.*, "Internet of Things: Wireless Sensor Networks," *Int. Electron. Commision*, no. December, pp. 1–78, 2014.

[11] S. P. Kumar, "003 - Sensor networks: Evolution, opportunities, and challenges," *Proc. IEEE*, vol. 91, no. 8, pp. 1247–1256, 2003.

[12] A. M. Thike, S. Lupin, R. Chakirov, and Y. Vagapov, "Topology Optimisation of Wireless Sensor Networks," *MATEC Web Conf.*, vol. 82, 2016.

[13] G. Deepika and P. Rajapirian, "Wireless sensor network in precision agriculture: A survey," *2016 Int. Conf. Emerg. Trends Eng. Technol. Sci.*, pp. 1–4, 2016.

[14] G. R. Mendez, M. A. M. Yunus, and S. C. Mukhopadhyay, "A WiFi based smart wireless sensor network for monitoring an agricultural environment," in *2012 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, 2012, pp. 2640–2645.

[15] D. Al Bashish, M. Braik, and S. Bani-Ahmad, "A framework for detection and classification of plant leaf and stem diseases," in *2010 International Conference on Signal and Image Processing*, 2010, pp. 113–118.

[16] L. Ruiz-Garcia, L. Lunadei, P. Barreiro, and I. Robla, "A Review of Wireless Sensor Technologies and Applications in Agriculture and Food Industry: State of the Art and Current Trends," *Sensors*, vol. 9, no. 6, pp. 4728–4750, 2009.

[17] S. e. Yoo, J. e. Kim, T. Kim, S. Ahn, J. Sung, and D. Kim, "A2S: Automated Agriculture System based on WSN," in *2007 IEEE International Symposium on Consumer Electronics*, 2007, pp. 1–5.

[18] G. H. E. L. de Lima, L. C. e Silva, and P. F. R. Neto, "WSN as a Tool for Supporting

Agriculture in the Precision Irrigation," in *2010 Sixth International Conference on Networking and Services*, 2010, pp. 137–142.

[19] X. Li, Y. Deng, and L. Ding, "Study on precision agriculture monitoring framework based on WSN," in *2008 2nd International Conference on Anti-counterfeiting, Security and Identification*, 2008, pp. 182–185.

[20] S. Verma, N. Chug, and D. V Gadre, "Wireless Sensor Network for Crop Field Monitoring," in *2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, 2010, pp. 207–211.

[21] K. N. Kumar, "Zigbee Wireless Sensor Network Technology Study for Paddy Crop Field Monitoring," no. Icvci, pp. 1–5, 2011.

[22] J. Hwang, C. Shin, and H. Yoe, "Study on an agricultural environment monitoring server system using Wireless Sensor Networks.," *Sensors (Basel).*, vol. 10, no. 12, pp. 11189–211, 2010.

[23] "Precision Agriculture | IOT Philippines Inc. | +63 (2) 621-6355." [Online]. Available: http://www.iotphils.com/solutions/precision-agriculture/#prettyPhoto.

[24] H. M. Jawad, R. Nordin, S. K. Gharghan, A. M. Jawad, and M. Ismail, "Energy-Efficient Wireless Sensor Networks for Precision Agriculture: A Review.," *Sensors (Basel).*, vol. 17, no. 8, Aug. 2017.

[25] "An improved key distribution and updating mechanism for low power wide area networks (LPWAN)."

[26] LoRa Alliance, "White Paper: A Technical Overview of Lora and Lorawan," 2015.

[27] A. Augustin, J. Yi, T. Clausen, and W. Townsley, "A Study of LoRa: Long Range & Low Power Networks for the Internet of Things," *Sensors*, vol. 16, no. 12, p. 1466, 2016.

[28] A. Springer, W. Gugler, M. Huemer, L. Reindl, C. C. W. Ruppel, and R. Weigel, "Spread spectrum communications using chirp signals," in *EUROCOMM 2000. Information Systems for Enhanced Public Safety and Security. IEEE/AFCEA*, 2000, pp. 166–170.

[29] "A DIY Low-cost LoRa gateway." [Online]. Available: http://cpham.perso.univ-pau.fr/LORA/RPIgateway.html. [Accessed: 07-May-2018].

[30] O. Rioul and J. Magossi, "On Shannon's Formula and Hartley's Rule: Beyond the Mathematical Coincidence," *Entropy*, vol. 16, no. 9, pp. 4892–4910, Sep. 2014.

[31] Semtech Corporation, "LoRa Modulation Basics," no. May, pp. 1–26, 2015.

[32] "Shannon-Hartley theorem." [Online]. Available: http://www.linfo.org/shannon-hartley_theorem.html. [Accessed: 07-May-2018].

[33] A. Ranganathan, B. Danev, A. Francillon, and S. Capkun, "Physical-Layer Attacks on Chirp-based Ranging Systems."

[34] "SX127x Low Power Long Range Transceiver - Semtech | Mouser Germany." [Online]. Available: https://www.mouser.de/new/semtech/semtech-sx1276-transceiver/. [Accessed: 07-May-2018].

[35] "[LoRa] LoRa Packet Structure 지돌이의 블로그 입니다!" [Online]. Available: http://ablog.jc-lab.net/107. [Accessed: 07-May-2018].

[36] "LoRaWan, a dedicated IoT network - Witekio." [Online]. Available: https://witekio.com/de/blog/lorawan-dedicated-iot-network/. [Accessed: 07-May-2018].

[37] *Journal of computer science.* Science Publications, 2005.

[38] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melià-Seguí, and T. Watteyne, "Understanding the Limits of LoRaWAN."

[39] "MSP430F5529 LaunchPad™ Development Kit (MSP‑EXP430F5529LP) User's Guide MSP430F5529 LaunchPad™ Development Kit (MSP‑‑EXP430F5529LP) Figure 1. MSP430F5529 LaunchPad Development Kit," 2013.

[40] "Grove - Barometer Sensor(BME280)." [Online]. Available:

http://wiki.seeedstudio.com/Grove-Barometer_Sensor-BME280/. [Accessed: 07-May-2018].

[41] “Dragino LoRa Shield - support 868M frequency - Arduino & Compatible - Seeed Studio.” [Online]. Available: https://www.seeedstudio.com/s/Dragino-LoRa-Shield-support-868M-frequency-p-2651.html. [Accessed: 07-May-2018].

[42] “Raspberry Pi 2 Model B - Raspberry Pi.” [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/. [Accessed: 07-May-2018].

[43] “Raspberry Pi LoRa/GPS HAT - support 868M frequency - Hats & Plates - Seeed Studio.” [Online]. Available: https://www.seeedstudio.com/Raspberry-Pi-LoRa%2FGPS-HAT-support-868M-frequency-p-2695.html. [Accessed: 07-May-2018].

[44] “CCSTUDIO Code Composer Studio (CCS) Integrated Development Environment (IDE) | TI.com.” [Online]. Available: http://www.ti.com/tool/CCSTUDIO. [Accessed: 07-May-2018].

# Appendix 1 – Source Code

## 1. LoRa.h

```
/*
 * LORA.h
 *
 */

#ifndef LORA_H_
#define LORA_H_
#include <msp430.h>
#include <stdint.h>
#include <stdio.h>
/*
 * REGISTER ADDRESSES
 * */

#define RegFifo                    0x00
#define RegOpMode                  0x01
#define RegBitrateMsb              0x02
#define RegBitrateLsb              0x03
#define RegFdevMsb                 0x04
#define RegFdevLsb                 0x05
#define RegFrfMsb                  0x06
#define RegFrfMid                  0x07
#define RegFrfLsb                  0x08
#define RegPaConfig                0x09
#define RegPaRamp                  0x0A
#define RegOcp                     0x0B
#define RegLna                     0x0c
#define RegFifoAddrPtr             0x0d
#define RegFifoTxBaseAddr          0x0e
#define RegFifoRxBaseAddr          0x0f
#define REG_FIFO_RX_CURRENT_ADDR 0x10
#define RegIrqFlags                0x12
#define REG_RX_NB_BYTES            0x13
#define REG_PKT_SNR_VALUE          0x19
#define REG_PKT_RSSI_VALUE         0x1a
#define RegRssiValueLora           0x1b
#define RegModemCfg1               0x1d
```

```
#define RegModemCfg2            0x1e
#define RegPreambleMsb          0x20
#define RegPreambleLsb          0x21
#define RegPayloadLength        0x22
#define RegModemCfg3            0x26
#define REG_FREQ_ERROR_MSB      0x28
#define REG_FREQ_ERROR_MID      0x29
#define REG_FREQ_ERROR_LSB      0x2a
#define REG_RSSI_WIDEBAND       0x2c
#define RegDetectionOptimize    0x31
#define RegNodeAddr             0x33
#define RegBroadcastAddr        0x34
#define RegImageCal             0x3B
#define RegIrqFlags1            0x3E
#define RegIrqFlags2            0x3F
#define RegDetectionThreshold   0x37
#define RegSyncWord             0x39
#define REG_DIO_MAPPING_1       0x40
#define RegVersion              0x42

// modes
#define ModeLongRange           0x80
#define ModeSleep               0x00
#define ModeStandby             0x01
#define ModeTx                  0x03
#define MODE_RX_CONTINUOUS      0x05
#define MODE_RX_SINGLE          0x06
#define ModeLoraStandby         0x81
#define ModeLoraCad             0x87
// PA config
#define LoRa_PA_Boost           0x80

// IRQ masks
#define IrqMaskTxDone             0x08
#define IRQ_PAYLOAD_CRC_ERROR_MASK 0x20
#define IRQ_RX_DONE_MASK          0x40

#define MaxPacketLength           255

#define RfImagecalMask                  0xBF
#define RfImagecalStart                 0x40

#define RfImagecalRunning               0x20
#define RfImagecalDone                  0x00  // Default

#define HEADER_ON                  0
#define HEADER_OFF                 1
```

```
#define CRC_ON                      1
#define CRC_OFF                     0
#define LORA                        1
#define FSK                         0
#define BroadcastAddr              0x00
#define MAX_LENGTH                 255
#define MAX_PAYLOAD                251
#define MAX_LENGTH_FSK              64
#define MaxPayloadFsk             60

//LORA CODING RATE:
#define CR_5                        0x01
#define CR_6                        0x02
#define CR_7                        0x03
#define CR_8                        0x04

//LORA SPREADING FACTOR:
#define SF_6                        0x06
#define SF_7                        0x07
#define SF_8                        0x08
#define SF_9                        0x09
#define SF_10                       0x0A
#define SF_11                       0x0B
#define SF_12                       0x0C

// LORA Bandwidths
#define BW_7_8                      0x00
#define BW_10_4                     0x01
#define BW_15_6                     0x02
#define BW_20_8                     0x03
#define BW_31_25                    0x04
#define BW_41_7                     0x05
#define BW_62_5                     0x06
#define BW_125                      0x07
#define BW_250                      0x08
#define BW_500                      0x09
// end


//LORA MODES:
#define LORA_SLEEP_MODE             0x80
#define LORA_STANDBY_MODE           0x81
#define LoraTxMode                  0x83
#define LORA_RX_MODE                0x85
#define LoraStandbyFskRegMode       0xC1
```

```
#define CH_10_868                0xD84CCC // channel 10, central freq
= 865.20MHz, = 865200000*RH_LORA_FCONVERT


//FSK MODES:
#define FSK_SLEEP_MODE              0x00
#define ModeFskStandby              0x01
#define FSK_TX_MODE                 0x03
#define FSK_RX_MODE                 0x05

#define RssiOffset                  139

#define Packet_Data                 0x10
#define Packet_Ack                  0x20
#define Packet_Flag_Ack             0x08


/*Structures*/

struct pack
{

    // Structure Variable : Packet destination
    uint8_t dst;
    // Structure Variable : Packet type
    uint8_t type;
    // Structure Variable : Packet source
    uint8_t src;
    // Structure Variable : Packet number
    uint8_t packnum;
    // Structure Variable : Packet length
    uint8_t length;
    // Structure Variable : Packet payload
    char* data;
    // Structure Variable : Retry number
    uint8_t retry;
};

/*Prototypes*/
void SPI_Initialize();
void SPI_Send_Byte(uint8_t Data);
uint8_t SPI_Receive_Byte();
void DELAY_Ms(uint16_t ms);
void DELAY_Sec(uint8_t sec);
void LORA_Init();
uint8_t LORA_Read_Register(uint8_t reg);
void LORA_Write_Register(uint8_t reg, uint8_t value);
```

```
void LORA_Set_Frequency(uint32_t frequency);
void LORA_Set_Power(int8_t dbm,uint8_t OP);
void LORA_Set_Power_Db(int8_t dbm);
void LORA_Set_Spreading_Factor(uint8_t spFactor);
void LORA_Set_Signal_BW(uint16_t BW);
void LORA_Set_Coding_Rate(uint16_t d);
void LORA_Set_Preamble_Length(uint32_t preamble_length);
void LORA_Set_Sync_Word(uint8_t sync_word);
void LORA_Set_CRC(uint8_t crc);
void LORA_Clear_Flags();
void LORA_Standby_Mode();
void LORA_Begin_Tranmission(uint8_t header);
void LORA_End_Transmission();
void LORA_Implicit_Header();
void LORA_Explicit_Header();
size_t LORA_Write(const uint8_t *buffer, size_t size);
size_t LORA_Write_Byte(uint8_t byte);
void LORA_Rx_Chain_Cal();
void LORA_Set_Lora_Mode();
void LORA_Set_Mode(uint8_t libelium_mode);
void LORA_Set_Node_Address(uint8_t addr);
void LORA_Carrier_Sense();
uint8_t LORA_CAD(uint8_t count);
uint8_t LORA_Get_Spreading_Factor();
void LORA_Set_Packet_Type(uint8_t type);

void LORA_Dump_Registers();
void LORA_Send_Packet(uint8_t dest, char *payload,uint8_t length);
void LORA_Set_Max_Current(uint8_t current_rate);
#endif /* LORA_H_ */
```

## 2. LoRa.c

```
/*
 * LORA.c
 *
 */
#include "LORA.h"
#define LoRa_Frequency          868000000
#define LoRa_PA_Boost_Pin           1
#define LoRa_Output_RFO_Pin      0

/*Global Variables*/
volatile uint8_t implicit_header_mode=0;
volatile uint8_t modem=0;
volatile uint8_t node_address=0;
```

```
volatile uint8_t bandwidth = BW_125;
volatile uint8_t RSSI=0;
volatile uint8_t requestAck=0;
volatile uint8_t destination=0;
volatile uint8_t packetNumber=0;
volatile uint8_t payloadlength=0;
struct pack packet_sent;
/*
 * Initialize SPI USCI and Reset pin of LORA
 * UCA0
 * P2.0 - NSS
 * P3.4 – UCA0SOMI
 * P3.3 – UCA0SIMO
 * P2.7 – UCA0CLK
 * P1.5 - RESET
 * */
void SPI_Initialize(){
    /*Make Pin 1.5 Output*/
    P1OUT |= BIT5;
    P1DIR |= BIT5;
    /*Make Pin 2.0 Output For NSS*/
    P2OUT |= BIT0;
    P2DIR |= BIT0;

    P3SEL |= BIT3+BIT4;                         // P3.3,4 option select
    P2SEL |= BIT7;
    UCA0CTL1 = UCSWRST;
    UCA0CTL0 |= UCCKPH + UCMSB + UCMST + UCSYNC; // 3-pin, 8-bit SPI
master
    UCA0CTL1 |= UCSSEL_2; // SMCLK
    UCA0BR0 |= 0x02; // /2
    UCA0BR1 = 0; //
    UCA0MCTL = 0; // No modulation
    UCA0CTL1 &= ~UCSWRST; // **Initialize USCI state machine**
}

/*
 * Milisecond Delay using intrinsic delay function
 * For 8MHz Clock speed
 * */
void DELAY_Ms(uint16_t ms){
    while(ms>0){
        __delay_cycles(4000);
        ms--;
    }
}
```

```
/*
 * Second Delay using intrinsic delay function
 * For 8MHz Clock speed
 * */
void DELAY_Sec(uint8_t sec){
    while(sec>0){
       DELAY_Ms(1000);
       sec--;
    }
}

/*
 * Function to Send Byte Data to Slave over SPI
 * */
void SPI_Send_Byte(uint8_t Data){
    while (!(UCA0IFG & UCTXIFG)); // USCI_A0 TX buffer ready?
    UCA0TXBUF = Data; // Send 0xAA over SPI to Slave
    while (!(UCA0IFG & UCTXIFG));
    //DELAY_Ms(1);
}

/*
 * Function to Receive Byte Data from Slave over SPI
 * */
uint8_t SPI_Receive_Byte(){
    while (!(UCA0IFG & UCRXIFG)); // USCI_A0 RX Received?
    uint8_t data=0;
    data = UCA0RXBUF; // Store received data
    return data;
}


/*
 * LoRa Module Initialization Function
 * */
void LORA_Init(){
    SPI_Initialize();
    /*Set NSS High*/
    P2OUT |= BIT0; // NSS High
    /*Cycle Reset of LoRa module*/
    P1OUT &= (~BIT5); // RESET low
    DELAY_Ms(10);
    P1OUT |= BIT5; // RESET High
    DELAY_Ms(10);
    /*Get LoRa Module Version*/
    uint8_t module_version=0;
    module_version = LORA_Read_Register(RegVersion);
```

```
    /*Check if version is valid*/
    if (module_version != 0x12)
        while(1);
    /*Calibrate Rx Chain*/
    LORA_Rx_Chain_Cal();
    /*Set Max Current to 100mA*/
    LORA_Set_Max_Current(0x1B);
    /*Put Module in LoRa Mode*/
    LORA_Set_Lora_Mode();
    /*Set CRC On*/
    LORA_Set_CRC(1);
    /*Set Default sync word for non LoRaWAN*/
    LORA_Set_Sync_Word(0x12);
}

/*
 * Read the value of a specific register from
 * LoRa Module
 * */
uint8_t LORA_Read_Register(uint8_t reg){
    /*Translate Address*/
    uint8_t addr=reg & 0x7f;
    /*Set NSS Low*/
    P2OUT &= (~BIT0);
    SPI_Send_Byte(addr);
    SPI_Send_Byte(0x00);
    uint8_t value;
    value = SPI_Receive_Byte();
    /*Set NSS High*/
    P2OUT |= (BIT0);
    return value;
}

/*
 * Write to the value of a specific register from
 * LoRa Module
 * */
void LORA_Write_Register(uint8_t reg, uint8_t value){
    /*Translate Address*/
    uint8_t addr=reg | 0x80;
    /*Set NSS Low*/
    P2OUT &= (~BIT0);
    SPI_Send_Byte(addr);
    SPI_Send_Byte(value);
    /*Set NSS High*/
    P2OUT |= (BIT0);
}
```

```
/*
 * Set frequency of the LoRa Module
 * */
void LORA_Set_Frequency(uint32_t frequency){
      uint8_t state;
      /*Save the current state*/
      state=LORA_Read_Register(RegOpMode);
      /*Check if modem is in LoRa Mode*/
      if(modem == LORA)
          LORA_Write_Register(RegOpMode, ModeLoraStandby);
      else
          LORA_Write_Register(RegOpMode, ModeFskStandby);
      uint32_t f,f1;
      f = ((frequency >> 16) & 0x0FF);     // frequency channel MSB
      LORA_Write_Register(RegFrfMsb, f);
      f = ((frequency >> 8) & 0x0FF);        // frequency channel MIB
      LORA_Write_Register(RegFrfMid, f);
      f = (frequency & 0xFF);                // frequency channel LSB
      LORA_Write_Register(RegFrfLsb, f);
      DELAY_Ms(100);
      /*verify*/
      f=LORA_Read_Register(RegFrfMsb);
      /*save MSB in f1*/
      f1=(f<<8) & 0xFFFFFF;
      /*retrieve mid bit*/
      f=LORA_Read_Register(RegFrfMid);
      /*save mid byte in f1 */
      f1= (f1<<8) + ((f<<8) & 0xffffff);
      /*retrieve lsb*/
      f=LORA_Read_Register(RegFrfLsb);
      /*save lsb in f1*/
      f1= (f1) + (f & 0xffffff);
      /*Loop here is frequencies dont match*/
      while(f1!=frequency);
      /*restore state*/
      LORA_Write_Register(RegOpMode,state);
      DELAY_Ms(100);

}


/*
 * Set Output power of the LoRa Module
 * */
void LORA_Set_Power(int8_t dbm,uint8_t OP){
    int8_t power;
```

```
    power = dbm;
    if (LoRa_Output_RFO_Pin == OP) {
       // RFO
       if (power < 0) {
           power = 0;
       } else if (power > 14) {
           power = 14;
       }

       LORA_Write_Register(RegPaConfig, 0x70 | power);
     } else {
       // PA BOOST
       if (power < 2) {
           power = 2;
       } else if (power > 17) {
           power = 17;
       }

       LORA_Write_Register(RegPaConfig, LoRa_PA_Boost | (power - 2));
     }
}

/*Set dBm only*/
void LORA_Set_Power_Db(int8_t dbm){
    uint8_t state,value;
    state=LORA_Read_Register(RegOpMode);
    LORA_Set_Lora_Mode();
    LORA_Write_Register(RegOpMode, ModeLoraStandby);
    uint8_t pmax=15;
    value = dbm-pmax + 15;
    value = value | 0b10000000;
    LORA_Write_Register(RegPaConfig, value);
    DELAY_Ms(10);
    while(value!=LORA_Read_Register(RegPaConfig));
    LORA_Write_Register(RegOpMode, state);
    DELAY_Ms(100);
}

/*
 * Set Spreading Factor of the LoRa Module
 * */
void LORA_Set_Spreading_Factor(uint8_t spFactor){
    int8_t config,config1;
    uint8_t state;
    state=LORA_Read_Register(RegOpMode);
    LORA_Set_Lora_Mode();
    LORA_Write_Register(RegOpMode, ModeLoraStandby);
```

```
    config=LORA_Read_Register(RegModemCfg2);
    switch(spFactor)
    {
    case SF_6:  config = config & 0b01101111;  // clears bits 7 & 4
from REG_MODEM_CONFIG2
    config = config | 0b01100000;  // sets bits 6 & 5 from
REG_MODEM_CONFIG2
    LORA_Implicit_Header();     // Mandatory headerOFF with SF = 6
    break;
    case SF_7:  config = config & 0b01111111;  // clears bits 7 from
REG_MODEM_CONFIG2
    config = config | 0b01110000;  // sets bits 6, 5 & 4
    break;
    case SF_8:  config = config & 0b10001111;  // clears bits 6, 5 & 4
from REG_MODEM_CONFIG2
    config = config | 0b10000000;  // sets bit 7 from
REG_MODEM_CONFIG2
    break;
    case SF_9:  config = config & 0b10011111;  // clears bits 6, 5 & 4
from REG_MODEM_CONFIG2
    config = config | 0b10010000;  // sets bits 7 & 4 from
REG_MODEM_CONFIG2
    break;
    case SF_10: config = config & 0b10101111;  // clears bits 6 & 4
from REG_MODEM_CONFIG2
    config = config | 0b10100000;  // sets bits 7 & 5 from
REG_MODEM_CONFIG2
    break;
    case SF_11: config = config & 0b10111111;  // clears bit 6 from
REG_MODEM_CONFIG2
    config = config | 0b10110000;  // sets bits 7, 5 & 4 from
REG_MODEM_CONFIG2
    //getBW();
    uint8_t bw;
    bw=LORA_Read_Register(RegModemCfg1)>>4;
    if( bw == BW_125)
    { // LowDataRateOptimize (Mandatory with SF_11 if BW_125)

        config1=LORA_Read_Register(RegModemCfg3);
        config1 = config1 | 0b00001000;
        LORA_Write_Register(RegModemCfg3,config1);
    }
    break;
    case SF_12: config = config & 0b11001111;  // clears bits 5 & 4
from REG_MODEM_CONFIG2
    config = config | 0b11000000;  // sets bits 7 & 6 from
REG_MODEM_CONFIG2
    if( bandwidth == BW_125)
```

```
    { // LowDataRateOptimize (Mandatory with SF_12 if BW_125)

        config1=LORA_Read_Register(RegModemCfg3);
        config1 = config1 | 0b00001000;
        LORA_Write_Register(RegModemCfg3,config1);

    }
    break;
    }

    // Check if it is neccesary to set special settings for SF=6
    if( spFactor == SF_6 )
    {
        // header OFF with SF=6 (Implicit mode)
        LORA_Implicit_Header();
        LORA_Write_Register(RegDetectionOptimize, 0xc5);
        LORA_Write_Register(RegDetectionThreshold, 0x0c);
    }
    else
    {
        LORA_Explicit_Header();
        LORA_Write_Register(RegDetectionOptimize, 0xc3);
        LORA_Write_Register(RegDetectionThreshold, 0x0a);
    }
    // set the AgcAutoOn in bit 2 of REG_MODEM_CONFIG3
    config1 = (LORA_Read_Register(RegModemCfg3));
    config1=config1 | 0b00000100;
    LORA_Write_Register(RegModemCfg3,config1);
    // write the new SF
    LORA_Write_Register(RegModemCfg2, config);      // Update config2
    DELAY_Ms(100);
    /*Check the update*/
    config1=LORA_Read_Register(RegModemCfg2);
    while(config1!=config);
    //Go back to previous state
    LORA_Write_Register(RegOpMode, state);

}

/*
 * Set Signal Bandwidth of LoRa module
 * */
void LORA_Set_Signal_BW(uint16_t BW){
    int8_t state;
    uint8_t config,config1;
    //uint16_t level;
    bandwidth=BW;
```

```
    /*Save current module state*/
    state=LORA_Read_Register(RegOpMode);
    LORA_Set_Lora_Mode();
    LORA_Write_Register(RegOpMode, ModeLoraStandby);
    config=LORA_Read_Register(RegModemCfg1);
    config = config & 0b00001111;  // clears bits 7 - 4 from
RegModemCfg1
    switch(BW)
    {
    case BW_125:
        // 0111
        config = config | 0b01110000;
        uint8_t spreading_factor;
        spreading_factor=LORA_Get_Spreading_Factor();
        if( spreading_factor == 11 || spreading_factor == 12)
        { // LowDataRateOptimize (Mandatory with BW_125 if SF_11 or
SF_12)
            config1=LORA_Read_Register(RegModemCfg3);
            config1 = config1 | 0b00001000;
            LORA_Write_Register(RegModemCfg3,config1);
        }
        break;
    case BW_250:
        // 1000
        config = config | 0b10000000;
        break;
    case BW_500:
        // 1001
        config = config | 0b10010000;
        break;
    }
    LORA_Write_Register(RegModemCfg1,config);       // Update config1
    DELAY_Ms(100);
    config1=LORA_Read_Register(RegModemCfg1);
    while(config1!=config);
    LORA_Write_Register(RegOpMode, state);    // Getting back to
previous status
    DELAY_Ms(100);
}


/*
 * Set LoRa module coding rate
 * */
void LORA_Set_Coding_Rate(uint16_t d){
    int8_t config,config1;
    uint8_t state;
```

```
    state=LORA_Read_Register(RegOpMode);
    LORA_Set_Lora_Mode();
    LORA_Write_Register(RegOpMode, ModeLoraStandby);
    config = LORA_Read_Register(RegModemCfg1);
    config = config & 0b11110001;  // clears bits 3 - 1 from
REG_MODEM_CONFIG1
    switch(d)
    {
        case CR_5:
             config = config | 0b00000010;
             break;
        case CR_6:
             config = config | 0b00000100;
             break;
        case CR_7:
             config = config | 0b00000110;
             break;
        case CR_8:
             config = config | 0b00001000;
             break;
     }
      LORA_Write_Register(RegModemCfg1, config);
      DELAY_Ms(100);
      config1 = LORA_Read_Register(RegModemCfg1);
      while(config!=config1);
      LORA_Write_Register(RegOpMode, state);
      DELAY_Ms(100);
}

/*
 * Set LoRa preamble length
 * */
void LORA_Set_Preamble_Length(uint32_t preamble_length){
    LORA_Write_Register(RegPreambleMsb, (uint8_t)(preamble_length >>
8));
    LORA_Write_Register(RegPreambleLsb, (uint8_t)(preamble_length >>
0));
}

/*
 *
 * Set Sync Word*/
void LORA_Set_Sync_Word(uint8_t sync_word){
    LORA_Write_Register(RegSyncWord, sync_word);
}

/*
```

```
 * Set CRC
 * */
void LORA_Set_CRC(uint8_t crc){
    if(crc){
        LORA_Write_Register(RegModemCfg2,
LORA_Read_Register(RegModemCfg2) | 0x04);
        while(!(LORA_Read_Register(RegModemCfg2) & 0x04));
    }
    else
        LORA_Write_Register(RegModemCfg2,
LORA_Read_Register(RegModemCfg2) & 0xfb);

}

/*
 * Clear the flags of LoRa Module*/
void LORA_Clear_Flags(){
    uint8_t status;
    status=LORA_Read_Register(RegOpMode);
    if(modem==LORA){
        // LoRa mode
        LORA_Write_Register(RegOpMode, ModeLoraStandby);  // Stdby
mode to write in registers
        LORA_Write_Register(RegIrqFlags, 0xFF); // LoRa mode flags
register
        LORA_Write_Register(RegOpMode, status);        // Getting back
to previous status
    }
    else{
        // FSK mode
        LORA_Write_Register(RegOpMode, ModeFskStandby);   // Stdby
mode to write in registers
        LORA_Write_Register(RegIrqFlags1, 0xFF); // FSK mode flags1
register
        LORA_Write_Register(RegIrqFlags2, 0xFF); // FSK mode flags2
register
        LORA_Write_Register(RegOpMode, status);        // Getting
    }
}
/*
 * Put LoRa Module in Standby Mode
 * */

void LORA_Standby_Mode(){
    LORA_Write_Register(RegOpMode, ModeLongRange | ModeStandby);
}

/*
```

```
 * Begin sending packets over LoRa
 * */
void LORA_Begin_Tranmission(uint8_t header){
    /*Put Module in Standby Mode*/
    LORA_Standby_Mode();
    if(header)
        LORA_Implicit_Header();
    else
        LORA_Explicit_Header();
    /*Reset payload length and buffer address*/
    LORA_Write_Register(RegFifoAddrPtr, 0);
    LORA_Write_Register(RegPayloadLength, 0);

}

/*
 * End Transmission over LoRa
 * */
void LORA_End_Transmission(){
    /*Put LoRa Module in Transmission Mode*/
    LORA_Write_Register(RegOpMode, ModeLongRange | ModeTx);
    /*Wait for Transmission to Complete*/
    while ((LORA_Read_Register(RegIrqFlags) & IrqMaskTxDone) == 0);
    /*Clear IRQs of the Module*/
    LORA_Write_Register(RegIrqFlags, IrqMaskTxDone);
}

/*
 * Put Module in implicit header mode
 * */
void LORA_Implicit_Header(){
    implicit_header_mode=1;
    LORA_Write_Register(RegModemCfg1, LORA_Read_Register(RegModemCfg1)
| 0x01);
}

/*
 * Put Module in explicit header mode
 * */
void LORA_Explicit_Header(){
    implicit_header_mode=0;
    LORA_Write_Register(RegModemCfg1, LORA_Read_Register(RegModemCfg1)
& 0xfe);
}

/*
 * Send Byte Data over LoRa
```

```
 * */
size_t LORA_Write(const uint8_t *buffer, size_t size){
    /*Get payload length*/
    uint16_t currentLength = LORA_Read_Register(RegPayloadLength);
    /*Check the size of payload*/
    if ((currentLength + size) > MaxPacketLength) {
        size = MaxPacketLength - currentLength;
    }
    /*Write payload to buffer*/
    size_t i ;
    for (i = 0; i < size; i++) {
        LORA_Write_Register(RegFifo, buffer[i]);
    }
    /*update payload length in the module register*/
    LORA_Write_Register(RegPayloadLength, currentLength + size);
    return size;
}


/*
 * Send single byte over LoRa
 * */
size_t LORA_Write_Byte(uint8_t byte){
    return LORA_Write(&byte, sizeof(byte));
}

/*
 *  Perform the Receiver chain calibration for LF and HF bands
 *  */
void LORA_Rx_Chain_Cal(){
    /* Cutoff the PA ,RFO output, power = -1 dBm*/
    LORA_Write_Register( RegPaConfig, 0x00 );

    /*Start Receiver chain calibration of LF band*/
    LORA_Write_Register( RegImageCal, ( LORA_Read_Register(
RegImageCal ) & RfImagecalMask ) | RfImagecalStart );
    while( ( LORA_Read_Register( RegImageCal ) & RfImagecalRunning )
== RfImagecalRunning );

    // Sets a Frequency in HF band
    LORA_Set_Frequency(0xD90000);//Channel 17 at 868MHz

    // Launch Rx chain calibration for HF band
    LORA_Write_Register( RegImageCal, ( LORA_Read_Register(
RegImageCal ) & RfImagecalMask ) | RfImagecalStart );
    while( ( LORA_Read_Register( RegImageCal ) & RfImagecalRunning )
== RfImagecalRunning );
```

```
}

/*
 * Put the Module in LoRa Mode
 * */
void LORA_Set_Lora_Mode(){
    uint8_t state;
    //i=0;
    do{
        DELAY_Ms(200);
        LORA_Write_Register(RegOpMode, ModeSleep);    // Sleep mode
(mandatory to set LoRa mode)
        LORA_Write_Register(RegOpMode, ModeLongRange);    // LoRa
sleep mode
        LORA_Write_Register(RegOpMode, ModeLoraStandby);
        //DELAY_Ms(50+i*10);
        state = LORA_Read_Register(RegOpMode);

        } while (state!=ModeLoraStandby);   // LoRa standby mode
    if(state==ModeLoraStandby)
        modem=LORA;
    else
        while(1);
}

/*
 * Set Lora Mode according to libelium documentation
 * Sets the proper badnwidth, spreading factor and
 * coding rate of LoRa Modulation
 *
 * */
void LORA_Set_Mode(uint8_t libelium_mode){
    int8_t state;
    state=LORA_Read_Register(RegOpMode);
    uint8_t config,config1;
    if(modem == FSK)
    LORA_Set_Lora_Mode();
    LORA_Write_Register(RegOpMode, ModeLoraStandby);
    switch (libelium_mode)
        {
        /* mode 1 (better reach, medium time on air)*/
        case 1:
            LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
            LORA_Set_Spreading_Factor(SF_12);       // SF = 12
            LORA_Set_Signal_BW(BW_125);      // BW = 125 KHz
            break;
```

```
/* mode 2 (medium reach, less time on air)*/
case 2:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_12);       // SF = 12
    LORA_Set_Signal_BW(BW_250);      // BW = 250 KHz
    break;

/* mode 3 (worst reach, less time on air)*/
case 3:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_10);       // SF = 10
    LORA_Set_Signal_BW(BW_125);      // BW = 125 KHz
    break;

/* mode 4 (better reach, low time on air)*/
case 4:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_12);       // SF = 12
    LORA_Set_Signal_BW(BW_500);      // BW = 500 KHz
    break;

/* mode 5 (better reach, medium time on air)*/
case 5:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_10);       // SF = 10
    LORA_Set_Signal_BW(BW_250);      // BW = 250 KHz
    break;

/* mode 6 (better reach, worst time-on-air)*/
case 6:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_11);       // SF = 11
    LORA_Set_Signal_BW(BW_500);      // BW = 500 KHz
    break;

/* mode 7 (medium-high reach, medium-low time-on-air)*/
case 7:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_9);        // SF = 9
    LORA_Set_Signal_BW(BW_250);      // BW = 250 KHz
    break;

/* mode 8 (medium reach, medium time-on-air)*/
case 8:
    LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
    LORA_Set_Spreading_Factor(SF_9);        // SF = 9
    LORA_Set_Signal_BW(BW_500);      // BW = 500 KHz
```

```
            break;

        /* mode 9 (medium-low reach, medium-high time-on-air)*/
        case 9:
            LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
            LORA_Set_Spreading_Factor(SF_8);        // SF = 8
            LORA_Set_Signal_BW(BW_500);      // BW = 500 KHz
            break;

        /* mode 10 (worst reach, less time_on_air)*/
        case 10:
            LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
            LORA_Set_Spreading_Factor(SF_7);        // SF = 7
            LORA_Set_Signal_BW(BW_500);      // BW = 500 KHz
            break;


        /* test for LoRaWAN channel*/
        case 11:
            LORA_Set_Coding_Rate(CR_5);        // CR = 4/5
            LORA_Set_Spreading_Factor(SF_12);        // SF = 12
            LORA_Set_Signal_BW(BW_125);      // BW = 125 KHz
            // set the sync word to the LoRaWAN sync word which is
0x34
            LORA_Set_Sync_Word(0x34);
            break;

        default:    while(1); // The indicated mode doesn't exist

        };
    if(libelium_mode==1){
        config=LORA_Read_Register(RegModemCfg1);
        config1=LORA_Read_Register(RegModemCfg2);
        while(!((config>>1)==0x39)||!((config1>>4)==SF_12));
    }
    LORA_Set_Sync_Word(0x12);
    LORA_Write_Register(RegOpMode, state);    // Getting back to
previous status
    DELAY_Ms(100);
}


/*
 * Set Address of the LoRa Node
 * Address should be from 1-255
 * Address one is usually the gateway
 *
```

```
 * */
void LORA_Set_Node_Address(uint8_t addr){
    node_address=addr;
    uint8_t status;
    status=LORA_Read_Register(RegOpMode);
    if(modem==LORA){
        /*Allow access to FSK registers in LoRa Mode*/
        LORA_Write_Register(RegOpMode,LoraStandbyFskRegMode);
    }
    else{
        /*Put module in standby mode*/
        LORA_Write_Register(RegOpMode,ModeFskStandby);
    }
    /* Write node and broadcast addresses to Registers*/
    LORA_Write_Register(RegNodeAddr, addr);
    LORA_Write_Register(RegBroadcastAddr, BroadcastAddr);
    DELAY_Ms(100);
    /*Restore Node status*/
    LORA_Write_Register(RegOpMode,status);
}

/*
 * Get Current Spreading Factor from the Module
 * */
uint8_t LORA_Get_Spreading_Factor(){
    //int8_t state;
    if(modem==FSK)
        while(1); /*Spreading factor not available in FSK mode*/
    return LORA_Read_Register(RegModemCfg2)>>4;
}
/*
 * Dump Registers of the LoRa Module
 * */
void LORA_Dump_Registers(){
    uint8_t i,x;
    for (i = 128; i==0 ; i--) {
        x= LORA_Read_Register(i);
        printf("\nRegister %x = %x",i,x);
      }

}
/*
 * Set Packet Type
 * */
void LORA_Set_Packet_Type(uint8_t type){
    packet_sent.type=type;
    if (type & Packet_Flag_Ack)
```

```
        requestAck=1;
}

/*
 * LoRa send packet
 * */
void LORA_Send_Packet(uint8_t dest, char *payload,uint8_t length){
    uint8_t state;
    state=LORA_Read_Register(RegOpMode);
    LORA_Clear_Flags();
    if (modem==LORA){
        LORA_Write_Register(RegOpMode, ModeLoraStandby);
    }
    else
        LORA_Write_Register(RegOpMode, ModeFskStandby);
    destination=dest;
    packet_sent.dst=dest;
    packet_sent.src=node_address;
    packet_sent.packnum=packetNumber;
    packetNumber++;
    //packet_sent.retries=0;
    payloadlength=length+4;
    if((modem==FSK)&& payloadlength>MaxPayloadFsk)
        payloadlength = MaxPayloadFsk;
    uint8_t i;
    //for(i = 0; i < payloadlength; i++){
        packet_sent.data = payload;   // Storing payload in packet
structure
    //}
    LORA_Write_Register(RegPayloadLength,payloadlength);
    uint8_t length_set=0;
    length_set=LORA_Read_Register(RegPayloadLength);
    while(length_set!=payloadlength);
    LORA_Write_Register(RegFifoAddrPtr,0x80);
    /*Write Packet to FIFO*/
    LORA_Write_Register(RegFifo,packet_sent.dst);
    LORA_Write_Register(RegFifo,packet_sent.type);
    LORA_Write_Register(RegFifo,packet_sent.src);
    LORA_Write_Register(RegFifo,packet_sent.packnum);
    /*Write Data*/
    for(i = 0; i < payloadlength; i++){
        LORA_Write_Register(RegFifo,payload[i]);
    }
    /*Restore Mode*/
    LORA_Write_Register(RegOpMode, state);
    /*Transmit...may call end tramit funciton*/
    LORA_Write_Register(RegOpMode, LoraTxMode);
```

```
    /*Wait for Transmission to Complete*/
    while ((LORA_Read_Register(RegIrqFlags) & IrqMaskTxDone) == 0);

    /*Clear IRQs of the Module*/
    LORA_Write_Register(RegIrqFlags, IrqMaskTxDone);
}
/*
 * Set Maximum Current of the module
 * */
void LORA_Set_Max_Current(uint8_t current_rate){
    int8_t state=0;
    current_rate|= 0xb00100000;
    state=LORA_Read_Register(RegOpMode);
    if(modem==LORA)
        LORA_Write_Register(RegOpMode,ModeLoraStandby);
    else
        LORA_Write_Register(RegOpMode,ModeFskStandby);

    LORA_Write_Register(RegOcp,current_rate);
    LORA_Write_Register(RegOpMode,state);
}
```

## 3. BME280.h

```
/*
 * BME280.h
 *
 */

#ifndef BME280_H_
#define BME280_H_
#include <msp430.h>
#include <stdint.h>
#include <stdio.h>
#define BME280                0x76 //I2C Address
#define MAX_BUFFER_SIZE     20

/*
 * Registers
 * */
#define        REG_T1              0x88
#define        REG_T2              0x8A
#define        REG_T3              0x8C

#define        REG_P1              0x8E
```

```
#define        REG_P2              0x90
#define        REG_P3              0x92
#define        REG_P4              0x94
 #define       REG_P5              0x96
#define        REG_P6              0x98
#define        REG_P7              0x9A
#define        REG_P8              0x9C
#define        REG_P9              0x9E

#define        REG_H1              0xA1
#define        REG_H2              0xE1
#define        REG_H3              0xE3
#define        REG_H4              0xE4
#define        REG_H5              0xE5
#define        REG_H6              0xE7

#define        REG_CHIPID             0xD0
#define        REG_VERSION            0xD1
#define        REG_SOFTRESET          0xE0

#define        REG_CAL26              0xE1  // R calibration stored
in 0xE1-0xF0

#define        REG_CONTROLHUMID       0xF2
#define        REG_STATUS             0XF3
#define        REG_CONTROL            0xF4
#define        REG_CONFIG             0xF5
#define        REG_PRESSUREDATA       0xF7
#define        REG_TEMPDATA           0xFA
#define        REG_HUMIDDATA          0xFD

//*****************************************************************
**********
// General I2C State Machine
************************************************
//*****************************************************************
**********

typedef enum I2C_ModeEnum{
    IDLE_MODE,
    NACK_MODE,
    TX_REG_ADDRESS_MODE,
    RX_REG_ADDRESS_MODE,
    TX_DATA_MODE,
    RX_DATA_MODE,
    SWITCH_TO_RX_MODE,
    SWITHC_TO_TX_MODE,
```

```
    TIMEOUT_MODE
} I2C_Mode;



/*=====================================================================
=====*/

/*=====================================================================
=====
    CALIBRATION DATA
    ---------------------------------------------------------------------
-----*/
struct calibration
    {
        uint16_t T1;
        int16_t  T2;
        int16_t  T3;

        uint16_t P1;
        int16_t  P2;
        int16_t  P3;
        int16_t  P4;
        int16_t  P5;
        int16_t  P6;
        int16_t  P7;
        int16_t  P8;
        int16_t  P9;

        uint8_t  H1;
        int16_t  H2;
        uint8_t  H3;
        int16_t  H4;
        int16_t  H5;
        int8_t   H6;
    };
/*=====================================================================
=====*/
void I2C_Init();
I2C_Mode I2C_Read(uint8_t reg, uint8_t count);
void CopyArray(uint8_t *source, uint8_t *dest, uint8_t count);
void BME280_Init();
I2C_Mode I2C_Write(uint8_t reg, uint8_t *data,uint8_t count);
uint8_t BME280_Calibrating();
void BME280_ReadFactory();
void BME280_Set_Defaults();
float BME280_Read_Temperature(void);
```

```
float BME280_Read_Pressure(void);
float BME280_Read_Humidity();
#endif /* BME280_H_ */
```

## 4. BME280.c

```
/*
 * BME280.c
 */
#include "BME280.h"
#include "LORA.h"

/* Used to track the state of the software state machine*/
I2C_Mode MasterMode = IDLE_MODE;


/* ReceiveBuffer: Buffer used to receive data in the ISR
 * RXByteCtr: Number of bytes left to receive
 * ReceiveIndex: The index of the next byte to be received in
ReceiveBuffer
 * TransmitBuffer: Buffer used to transmit data in the ISR
 * TXByteCtr: Number of bytes left to transfer
 * TransmitIndex: The index of the next byte to be transmitted in
TransmitBuffer
 * */
uint8_t ReceiveBuffer[MAX_BUFFER_SIZE] = {0};
uint8_t RXByteCtr = 0;
uint8_t ReceiveIndex = 0;
uint8_t TransmitBuffer[MAX_BUFFER_SIZE] = {0};
uint8_t TXByteCtr = 0;
uint8_t TransmitIndex = 0;
uint8_t message_tx[10]={0};
/* I2C Write and Read Functions */



/* The Register Address/Command to use*/
uint8_t TransmitRegAddr = 0;
struct calibration  factory_cal;
/*Hold raw temperature data*/
int32_t   temp_fine;



void I2C_Init();
```

```
/*
 * SDA P3.0
 * SCL P3.1
 * */

void I2C_Init(){
    //P3REN |= BIT0+BIT1;//Enable Pullups
    //P3OUT |= BIT0+BIT1;//Select Output type
    P3SEL |= BIT0+BIT1;//// Assign I2C pins to USCI_B0
    UCB0CTL1 |= UCSWRST;                      // Enable SW reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;     // I2C Master,
synchronous mode
    UCB0CTL1 = UCSSEL_2 + UCSWRST;            // Use SMCLK, keep SW
reset
    UCB0BR0 = 160;                            // fSCL = SMCLK/160 =
~100kHz
    UCB0BR1 = 0;
    UCB0I2CSA = BME280;                     // Slave Address is 048h
    UCB0CTL1 &= ~UCSWRST;                     // Clear SW reset,
resume operation
    UCB0IE |= UCNACKIE;

}


/*
 * Array Copy
 * */
void CopyArray(uint8_t *source, uint8_t *dest, uint8_t count)
{
    uint8_t copyIndex = 0;
    for (copyIndex = 0; copyIndex < count; copyIndex++)
    {
        dest[copyIndex] = source[copyIndex];
    }
}

/*
 * Read
 * */
I2C_Mode I2C_Read(uint8_t reg, uint8_t count){

    /* Initialize state machine */
        MasterMode = TX_REG_ADDRESS_MODE;
        TransmitRegAddr = reg;
        RXByteCtr = count;
        TXByteCtr = 0;
```

```
        ReceiveIndex = 0;
        TransmitIndex = 0;

        /* Initialize slave address and interrupts */
        UCB0I2CSA = BME280;
        UCB0IFG &= ~(UCTXIFG + UCRXIFG);       // Clear any pending
interrupts
        UCB0IE &= ~UCRXIE;                       // Disable RX
interrupt
        UCB0IE |= UCTXIE;                        // Enable TX
interrupt

        UCB0CTL1 |= UCTR + UCTXSTT;             // I2C TX, start
condition
        __bis_SR_register(LPM0_bits + GIE);              // Enter LPM0
w/ interrupts

        return MasterMode;
}


/*
 * Write
 * */
I2C_Mode I2C_Write(uint8_t reg, uint8_t *data,uint8_t count){

    /* Initialize state machine */
    MasterMode = TX_REG_ADDRESS_MODE;
    TransmitRegAddr = reg;
    //Copy register data to TransmitBuffer
    CopyArray(data, TransmitBuffer, count);
    TXByteCtr = count;
     RXByteCtr = 0;
     ReceiveIndex = 0;
     TransmitIndex = 0;

    /* Initialize slave address and interrupts */
    UCB0I2CSA = BME280;
    UCB0IFG &= ~(UCTXIFG + UCRXIFG);       // Clear any pending
interrupts
    UCB0IE &= ~UCRXIE;                       // Disable RX interrupt
    UCB0IE |= UCTXIE;                        // Enable TX interrupt

    UCB0CTL1 |= UCTR + UCTXSTT;             // I2C TX, start condition
    __bis_SR_register(LPM0_bits + GIE);              // Enter LPM0 w/
interrupts

    return MasterMode;
```

```
}
//*********************************************************************
**********
// I2C Interrupt
************************************************************
//*********************************************************************
**********

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCI_B0_VECTOR
__interrupt void USCI_B0_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(USCI_B0_VECTOR))) USCI_B0_ISR (void)
#else
#error Compiler not supported!
#endif
{
  //Must read from UCB0RXBUF
  uint8_t rx_val = 0;

  switch(__even_in_range(UCB0IV,0xC))
  {
    case USCI_NONE:break;                               // Vector 0 - no
interrupt
    case USCI_I2C_UCALIFG:break;                        // Interrupt
Vector: I2C Mode: UCALIFG
    case USCI_I2C_UCNACKIFG:break;                      // Interrupt
Vector: I2C Mode: UCNACKIFG
    case USCI_I2C_UCSTTIFG:break;                       // Interrupt
Vector: I2C Mode: UCSTTIFG
    case USCI_I2C_UCSTPIFG:break;                       // Interrupt
Vector: I2C Mode: UCSTPIFG
    case USCI_I2C_UCRXIFG:
        rx_val = UCB0RXBUF;
        if (RXByteCtr)
        {
          ReceiveBuffer[ReceiveIndex++] = rx_val;
          RXByteCtr--;
        }

        if (RXByteCtr == 1)
        {
          UCB0CTL1 |= UCTXSTP;
        }
        else if (RXByteCtr == 0)
        {
          UCB0IE &= ~UCRXIE;
          MasterMode = IDLE_MODE;
```

```
          __bic_SR_register_on_exit(CPUOFF);      // Exit LPM0
        }
        break;                      // Interrupt Vector: I2C Mode:
UCRXIFG
    case USCI_I2C_UCTXIFG:
        switch (MasterMode)
        {
          case TX_REG_ADDRESS_MODE:
              UCB0TXBUF = TransmitRegAddr;
              if (RXByteCtr)
                  MasterMode = SWITCH_TO_RX_MODE;   // Need to start
receiving now
              else
                  MasterMode = TX_DATA_MODE;        // Continue to
transmission with the data in Transmit Buffer
              break;

          case SWITCH_TO_RX_MODE:
              UCB0IE |= UCRXIE;              // Enable RX interrupt
              UCB0IE &= ~UCTXIE;             // Disable TX interrupt
              UCB0CTL1 &= ~UCTR;             // Switch to receiver
              MasterMode = RX_DATA_MODE;    // State state is to
receive data
              UCB0CTL1 |= UCTXSTT;          // Send repeated start
              if (RXByteCtr == 1)
              {
                  //Must send stop since this is the N-1 byte
                  while((UCB0CTL1 & UCTXSTT));
                  UCB0CTL1 |= UCTXSTP;      // Send stop condition
              }
              break;

          case TX_DATA_MODE:
              if (TXByteCtr)
              {
                  UCB0TXBUF = TransmitBuffer[TransmitIndex++];
                  TXByteCtr--;
              }
              else
              {
                  //Done with transmission
                  UCB0CTL1 |= UCTXSTP;     // Send stop condition
                  MasterMode = IDLE_MODE;
                  UCB0IE &= ~UCTXIE;                       // disable
TX interrupt
                  __bic_SR_register_on_exit(CPUOFF);      // Exit LPM0
              }
```

```
            break;

          default:
              __no_operation();
              break;
        }
        break;                      // Interrupt Vector: I2C Mode:
UCTXIFG
    default: break;
  }
}
/*
 * Init Sensor
 * */
void BME280_Init(){
    I2C_Init();
    /*verify Chip ID*/
    I2C_Read(REG_CHIPID, 1);
    /*Loop if incorrect*/
    while(ReceiveBuffer[0]!= 0x60);
    /*Reset Buffer index*/
    ReceiveIndex=0;
    /*Issue Soft Reset*/
    message_tx[0]=0xB6;
    I2C_Write(REG_SOFTRESET, message_tx,1);
    /*Wait for chip to be ready*/
    DELAY_Ms(300);
    /*Is it ready*/
    while(BME280_Calibrating())
    DELAY_Ms(100);
    BME280_ReadFactory();
    BME280_Set_Defaults();
    DELAY_Ms(100);

}


/*
 * Check if chip busy calibrating
 * */
uint8_t BME280_Calibrating(){
    uint8_t status;
    I2C_Read(REG_STATUS,1);
    status= ReceiveBuffer[0];
    ReceiveIndex=0;
    return (status & (1 << 0)) != 0;
}
```

```
/*
 * Read factory calibration and populate the variables
 * */
void BME280_ReadFactory(){
        uint16_t temp;
        //int16_t temp;
        I2C_Read(REG_T1,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.T1=temp;
        ReceiveIndex=0;
        I2C_Read(REG_T2,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.T2=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_T3,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.T3=(int16_t)temp;
        ReceiveIndex=0;


        I2C_Read(REG_P1,2);
        factory_cal.P1 = ((uint16_t)ReceiveBuffer[0]<<8) |
ReceiveBuffer[1];
        factory_cal.P1=(factory_cal.P1 >> 8) | (factory_cal.P1 << 8);
        ReceiveIndex=0;
        I2C_Read(REG_P2,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P2=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_P3,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P3=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_P4,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P4=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_P5,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P5=(int16_t)temp;
```

```
        ReceiveIndex=0;
        I2C_Read(REG_P6,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P6=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_P7,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P7=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_P8,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P8=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_P9,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.P9=(int16_t)temp;
        ReceiveIndex=0;

        I2C_Read(REG_H1,1);
        factory_cal.H1=ReceiveBuffer[0];
        ReceiveIndex=0;
        I2C_Read(REG_H2,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<8) | ReceiveBuffer[1];
        temp=(temp >> 8) | (temp << 8);
        factory_cal.H2=(int16_t)temp;
        ReceiveIndex=0;
        I2C_Read(REG_H3,1);
        factory_cal.H3=ReceiveBuffer[0];
        ReceiveIndex=0;
        I2C_Read(REG_H4,2);
        temp = ((uint16_t)ReceiveBuffer[0]<<4) | (ReceiveBuffer[1]&
0xF);
        factory_cal.H4=temp;
        ReceiveIndex=0;
        I2C_Read(REG_H5,2);
        temp = ((uint16_t)ReceiveBuffer[1]<<4) |
((uint16_t)ReceiveBuffer[0]>>4);
        factory_cal.H5=temp;
        ReceiveIndex=0;
        I2C_Read(REG_H6,1);
        factory_cal.H6=(int8_t)ReceiveBuffer[0];
        ReceiveIndex=0;
```

```
}


/*
 * Setup sensor with default values
 * */
void BME280_Set_Defaults(){
    // you must make sure to also set REGISTER_CONTROL after setting
the
        // CONTROLHUMID register, otherwise the values won't be
applied (see DS 5.4.3)
    message_tx[0]=0b101;
    I2C_Write(REG_CONTROLHUMID,message_tx, 1);
    message_tx[0]=0;
    I2C_Write(REG_CONFIG,message_tx,1);
    message_tx[0]=0b10110111;
    I2C_Write(REG_CONTROL,message_tx,1);
}


/*
 * Read Temperature from the sensor
 * */
float BME280_Read_Temperature(void){
    int32_t x1, x2;

    int32_t ADC_Temperature;
    I2C_Read(REG_TEMPDATA, 3);
    ADC_Temperature=(int32_t)ReceiveBuffer[0]<<16 |
(int32_t)ReceiveBuffer[1]<<8 | (int32_t)ReceiveBuffer[2];
    ReceiveIndex=0;


    if (ADC_Temperature == 0x800000) // value in case temp measurement
was disabled
        return 0;
    ADC_Temperature >>= 4;

    x1 = ((((ADC_Temperature>>3) - ((int32_t)factory_cal.T1 <<1))) *
            ((int32_t)factory_cal.T2)) >> 11;

    x2 = (((((ADC_Temperature>>4) - ((int32_t)factory_cal.T1)) *
              ((ADC_Temperature>>4) - ((int32_t)factory_cal.T1))) >>
12) *
            ((int32_t)factory_cal.T3)) >> 14;

    temp_fine = x1 + x2;
```

```
    float T = (temp_fine * 5 + 128) >> 8;
    return T/100;
}

/*
 * Read Pressure from the sensor
 * */
float BME280_Read_Pressure(void){
    int64_t x1, x2, pressure;

    BME280_Read_Temperature(); // must be done first to get temp_fine

    int32_t ADC_Pressure;
    I2C_Read(REG_PRESSUREDATA, 3);
    ADC_Pressure=(int32_t)ReceiveBuffer[0]<<16 |
(int32_t)ReceiveBuffer[1]<<8 | (int32_t)ReceiveBuffer[2];
    ReceiveIndex=0;
    if (ADC_Pressure == 0x800000) // value in case pressure
measurement was disabled
        return 0;
    ADC_Pressure >>= 4;

    x1 = ((int64_t)temp_fine) - 128000;
    x2 = x1 * x1 * (int64_t)factory_cal.P6;
    x2 = x2 + ((x1*(int64_t)factory_cal.P5)<<17);
    x2 = x2 + (((int64_t)factory_cal.P4)<<35);
    x1 = ((x1 * x1 * (int64_t)factory_cal.P3)>>8) +
          ((x1 * (int64_t)factory_cal.P2)<<12);
    x1 = (((((int64_t)1)<<47)+x1))*((int64_t)factory_cal.P1)>>33;

    if (x1 == 0) {
        return 0; // avoid exception caused by division by zero
    }
    pressure = 1048576 - ADC_Pressure;
    pressure = (((pressure<<31) - x2)*3125) / x1;
    x1 = (((int64_t)factory_cal.P9) * (pressure>>13) * (pressure>>13))
>> 25;
    x2 = (((int64_t)factory_cal.P8) * pressure) >> 19;

    pressure = ((pressure + x1 + x2) >> 8) +
(((int64_t)factory_cal.P7)<<4);
    return (float)pressure/(1013.25*256);
}


/*
```

```
 * Read Humidity value from the sensor
 * */
float BME280_Read_Humidity(){
    BME280_Read_Temperature(); // must be done first to get temp_fine

    int32_t ADC_Humidity;
    I2C_Read(REG_HUMIDDATA, 2);
    ADC_Humidity=(int32_t)ReceiveBuffer[0]<<8 |
(int32_t)ReceiveBuffer[1];
    ReceiveIndex=0;
    if (ADC_Humidity == 0x8000) // value in case humidity measurement
was disabled
        return 0;

    int32_t x1;

    x1 = (temp_fine - ((int32_t)76800));

    x1 = (((((ADC_Humidity << 14) - (((int32_t)factory_cal.H4) << 20)
-
            (((int32_t)factory_cal.H5) * x1)) + ((int32_t)16384)) >>
15) *
            (((((((x1 * ((int32_t)factory_cal.H6)) >> 10) *
                    (((x1 * ((int32_t)factory_cal.H3)) >> 11) +
((int32_t)32768))) >> 10) +
                    ((int32_t)2097152)) * ((int32_t)factory_cal.H2) +
8192) >> 14));

    x1 = (x1 - (((((x1 >> 15) * (x1 >> 15)) >> 7) *
            ((int32_t)factory_cal.H1)) >> 4));

    x1 = (x1 < 0) ? 0 : x1;
    x1 = (x1 > 419430400) ? 419430400 : x1;
    float humidity = (x1>>12);
    return  humidity / 1024.0;
}
```