IoT Labs – LoRa 2.4 GHz with sx1280

Table of Contents

Lab 0 (readings) : LoRa for ISM 433/868 MHz and 2.4 GHz....2
0.1 Introduction....2
0.2 LoRa in the 2.4 GHz Band....3
0.3 Path Loss Modeling....6
0.3.1 Free Space Environment....7
0.3.2. Indoor Environment....7
0.3.3. Urban Environment....8
0.4 Range versus Data Rate: Results....9
0.5 Discussion....10

Lab 1: Data transmission with LoRa 2.4GHz on ESP32 (Lolin D32) board and DevKit....12
1.1 The hardware....12
1.2 The software....13
1.3 Sending and receiving LoRa data frames with SX1280 modem....14
1.3.1 Transmitter code....14
1.3.1.1 Complete code of transmitter without interruption :....14
1.3.1.2 Transmitter code with interruption signal on RADIO_DIO1_PIN (33)....15
1.3.3 Receiver code....16
1.3.3.1 RadioLib SX128x Receive with Interrupts....16

Lab 2: LoRa 2.4 GHz LILYGO T-BEAM – communication....18
2.1 The hardware....18
2.2 The software....19
2.3 Sending and receiving LoRa data frames with SX1280 modem....23
2.3.1 Transmitter code....23
Complete code:....23
2.3.2 Receiver code....25

Lab 3: Long Distance Ranging with LoRa 2.4 GHz....28
3.1 Introduction....28
3.2 Ranging Operation....28
3.2.1 Radio Ranging Errors....29
3.2.1.1 Reference Oscillator Error....29
3.2.1.2 Analogue Group Delay Error....30
3.2.2 Channel Effects....31
3.2.2.1 Channel Selectivity & Multipath....31
3.2.2.2 Compensating Frequency Selectivity....33
3.2.2.3 Correcting Measurement Bias by Fingerprinting....34
3.3 Implementation: A Practical RTToF Measurement System....35
3.3.1. Introduction....35
3.3.2. The Ranging Protocol....35
3.3.3 The Channel Plan....35
3.3.4 Raw Results....36
3.3.5 Processing the Ranging Results....36
3.3.6. Before and After....38
3.4 Programming with simple configuration: Slave/Master....39
3.4.1 Ranging Master....39
3.4.2 Ranging Slave....41

IoT Labs – LoRa 2.4 GHz with sx1280

Lab 0

LoRa for ISM 433/868 MHz and 2.4 GHz

Recently, Semtech has released a Long Range (LoRa) chipset which operates at the globally available 2.4 GHz frequency band, on top of the existing sub-GHz, km-range offer, enabling hardware manufacturers to design region-independent chipsets.
The SX1280 LoRa module promises an ultra-long communication range while withstanding heavy interference in this widely used band.

In these labs, we first provide some mathematical description of the physical layer of LoRa in the 2.4 GHz band.

Free space, indoor and urban path loss models are used to simulate the propagation of the 2.4 GHz LoRa modulated signal at different spreading factors and bandwidths.
Additionally, we investigate the corresponding data rates. The results show a maximum range of 333 km in free space, 107 m in an indoor office-like environment and 867 m in an outdoor urban context. While a maximum data rate of 253.91 kbit/s can be achieved, the data rate at the longest possible range in every scenario equals 0.595 kbit/s.

Due to the configurable bandwidth and lower data rates, LoRa outperforms other technologies in the 2.4 GHz band in terms of communication range. In addition, both communication, ranging and localization applications deployed in private LoRa networks can benefit from the increased bandwidth and localization accuracy of this system when compared to public sub-GHz networks

In the second part we provide practical communication examples running on our Pomme-Pi Lora 2.4 GHz platform.

In the third part we present the principles of long distance ranging with LoRa 2.4 Ghz and SX1280 modem on our platform.

Finally we provide the practical coding examples to use with our platform.

0.1 Introduction

More than a decade ago, Long Range (LoRa) was invented to—as the name indicates—provide a low power wide area network (LPWAN) protocol operating at sub-GHz frequencies.
Because of local spectrum regulations, LoRa hardware modules need to be adapted to operate in different frequency bands. For example, in the US, the 915 MHz band is used, while in **Europe**, most LoRa chipsets operate in the **868 MHz band**. In the past, we have conducted extensive research on communication and localization with sub-GHz LPWAN.

Semtech, the founding member of the LoRa Alliance, recently released a LoRa chipset operating at **2.4 GHz**.

The move from sub-GHz to 2.4 GHz was mainly done in order to use the globally available 2.4 GHz Industrial, Scientific and Medical (ISM) band. This tackles the problem of having to develop multiple chipsets which operate at different frequency bands, thus paving the way for the development of a **universal chipset** which can operate **anywhere in the world**. This is especially valid for track and trace applications, where goods cross different zones worldwide.
Nevertheless, the potential benefits of LoRa in the 2.4 GHz band have not yet been investigated thoroughly, which constitutes the main objective of this paper.

In general, the range of sub-GHz LPWAN varies from a few kilometers in urban environments to more than 10 km in rural environments. Obviously, the maximum communication range between the transmitter and receiver also depends on the used frequency band. The goal of this research is to study the inverse relationship between the maximum communication range and the corresponding data rate of LoRa in the 2.4 GHz band.

Since sub-GHz LoRa devices are very limited in terms of their number of possible transmissions—i.e., to comply with **duty cycle regulations -** they are typically used for applications that do not require frequent communication.

Besides this, the long-range communication benefit of sub-GHz LoRa has been exploited to provide emergency services in GPS-less environments . Other application examples with sub-GHz LoRa include smart meter reading, environmental monitoring, smart farming and smart building applications.

On the other hand, LoRa devices that operate at 2.4 GHz are able to transmit at higher data rates because of the **higher available bandwidth**. Consequently, this technology can offer a balance for applications that require a higher data rate than LPWANs and a longer communication range than classic 2.4 GHz technologies such as Wi-Fi and Bluetooth.
Additionally, a higher bandwidth also allows for more accurate time-based localization. Thus, LoRa at 2.4 GHz represents an interesting solution for a variety of applications that involve indoor localization, such as warehouse management, but also for applications that require outdoor localization, such as construction site monitoring, livestock tracking, etc.
Moreover, the adoption of this technology can add flexibility to applications that require consistent **asset tracking** in both indoor and harsh outdoor environments; e.g., smart ports.

The main objectives of this introduction are the following:

- To provide an overview of the physical layer of LoRa operating at 2.4 GHz.
- To discuss the maximum communication range and data rate in three different scenarios: free space, indoor and urban environments.
- To discuss the impact of moving from LoRa at sub-GHz bands to 2.4 GHz on communication and localization applications.

The remainder of this part is structured as follows.

- A mathematical background of LoRa at 2.4 GHz is provided in the first section ().
- Next, three path loss models are presented in order to estimate the maximum communication range and corresponding data rate in a free space and in indoor and urban environments. Then the results of these estimations are shown and compared to other technologies operating in the 2.4 GHz band .
- Finally we discuss the impact on the application potential of LoRa at 2.4 GHz.

0.2 LoRa in the 2.4 GHz Band

The physical layer of LoRa is a proprietary and closed source. Therefore, there are no official references or protocol specifications for the transmitted RF signal , but enough information is available to illustrate the physical layer models of sub-GHz LoRa.

In this section, we modify the available physical layer models of sub-GHz LoRa to make them suitable for use in the 2.4 GHz frequency band.
Assume ***xs*(*k*)** is the transmitted **LoRa sample**; then, the received sampled signal ***xr*(*k*)** with index ***k*** can be expressed as

(1)
$$x_r(k) = a_r x_s(k-\tau)\, e^{i2\pi\Delta f k} + \omega(k),$$

where $ar<1$ is the received signal amplitude, τ is the time delay of the sample $xs(k)$ at the receiver, Δf is the frequency offset between the transmitter and the receiver, and $\omega(k)$ is the identically independently distributed (i.i.d.) complex-valued Gaussian noise with zero-mean and variance $\sigma 2$; i.e., $CN(0,\sigma 2)$.
The time and frequency synchronization are beyond the scope of this paper. Therefore, in the following, we will consider a simplified version of (1), shown in (2).

(2)
$$x_r(k) = a_r x_s(k) + \omega(k)$$

The LoRa standard linear **upchirp** - also called a base chirp - can be expressed as:

(3)
$$x_s(k) = e^{i2\pi\left(\frac{\mathrm{BW}}{2K}k^2 + f_\circ k\right)},$$

where **BW** is the operational bandwidth of the LoRa signal in the 2.4 GHz frequency band (as shown in Table 1) and $K = 2^{\mathrm{SF}}/\mathrm{BW}$ is the **symbol duration**, with **SF** representing the spreading factor (also shown in Table 1). Finally, $f_\circ$ is the initial frequency, which can be expressed as :

(4)
$$f_\circ = s\frac{\mathrm{BW}}{2^{\mathrm{SF}}},$$

where $s \in \{0, 1 \ldots 2^{\mathrm{SF}}\}$ is the transmitted data symbol. Setting $s = 0$ results in an upchirp, in which the frequency continuously increases during the symbol duration K.

Table 1
Parameters used for path loss modeling.

Model Parameter	Symbol	Value	Unit
Frequency	f	2.4	GHz
Spreading factor	SF	5–12	-
Bandwidth	BW	203/406/812/1625	kHz
Code rate	R_C	4/5	-
Transmission power	P_{TX}	12.5	dBm
Transmitter antenna gain	G_{TX}	2	dBi
Transmitter cable losses	L_{TX}	−2	dB
Fading margin	L_m	0	dB
Receiver antenna gain	G_{RX}	2	dBi
Receiver cable losses	L_{RX}	−2	dB
Base station height	h_b	20	m
Mobile station height	h_m	2	m

We can also present (3) as:

(5)
$$x_s(k) = W_K^{\frac{\mathrm{BW}}{2}k^2 + Kf_\circ k}, \quad W_K = e^{i2\pi/K}.$$

The model in (5) is the linearly cyclically shifted version of a base Zadoff–Chu (ZC) sequence . The ZC sequence possesses a unique autocorrelation property, in which the periodic autocorrelation is orthogonal (i.e., equal to zero) for all shifted replicas .
Therefore, the LoRa communication protocol uses this unique property to impose a random multiple access technique. Accordingly, an efficient utilization of the unlicensed spectrum can be obtained.

The correlation between the received signal and the base chirp leads to :

$$
\begin{aligned}
z(k) &= \frac{1}{K}\sum_{p=0}^{K-1} x\left(k+p\right) x_s^*(k)_{\text{mod } K} \\
&= \frac{1}{K}\sum_{p=0}^{K-1} (a_r x_s\left(k+p\right) + \omega\left(k+p\right)) x_s^*\left(k\right), \\
&= \begin{cases} a_r E_s + \nu_\omega & \text{for } p = 0 \\ \nu_\omega & \text{for } p \neq 0 \end{cases}
\end{aligned}
\tag{6}
$$

where Es is the energy of the symbol xs. Furthermore, $\nu\omega$ is the correlation between complex noise and the base chirp, which can be expressed as :

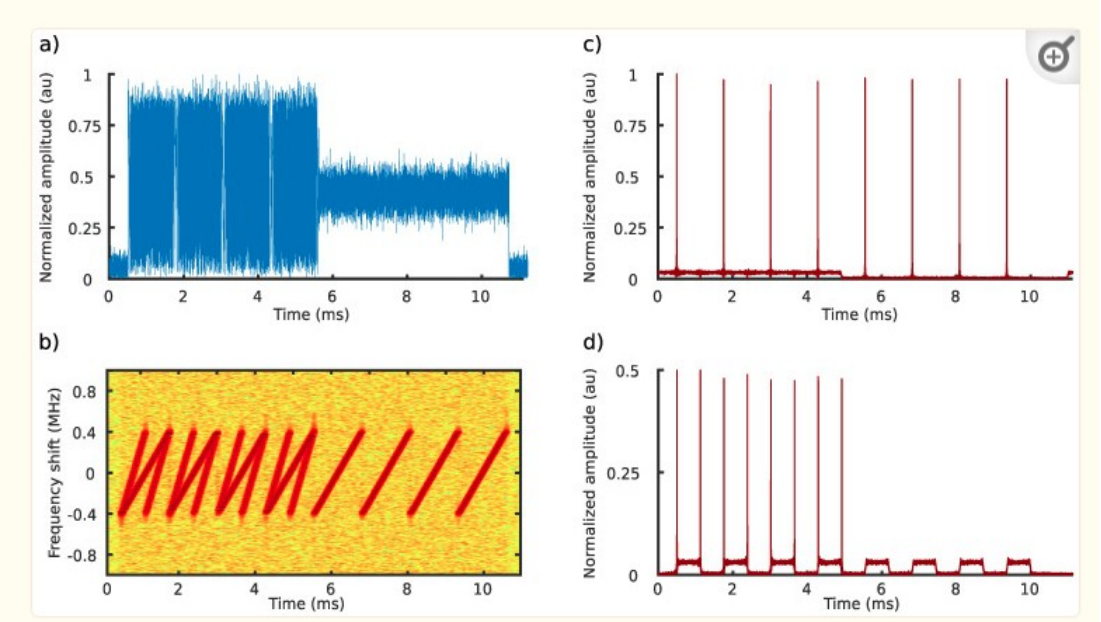
$$
\nu_\omega = \frac{1}{K}\sum_{p=0}^{K-1} x_s^*\left(k\right) \omega\left(k+p\right), \tag{7}
$$

in which $\nu\omega \sim CN(0, \sigma 2/K)$.

Figure 1 presents two received LoRa signals that constitute eight preamble (upchirp) symbols at 2.4 GHz with a bandwidth equal to 812 kHz. The short signal was transmitted at an SF of 9, while the longer signal was transmitted at an SF of 10.

Figure 1a,b shows the combined received signals in the time domain and in the spectrogram (i.e., time and frequency) domain, respectively.

Figure 1c,d shows the cross-correlation functions (6) when the received signals have been cross-correlated with base chirps of the SF equal to 10 and 9, respectively. It is clear that the two signals can be distinguished correctly, even though they interfere with each other. The unique orthogonality property of the ZC sequence allows the LoRa communication system to provide a multiple access technique in the 2.4 GHz frequency band.



Two received Long Range (LoRa) signals constitute eight preamble upchirp symbols at 2.4 GHz with a bandwidth equal to 812 kHz. The spreading factor (SF) of the short signal, which ended after approximately 5 ms, is equal to 9, while the SF of the long-duration signal is equal to 10. Figures (**a**) and (**b**) represent the combined received signals in the time domain and in the spectrogram (i.e., time and frequency) domain, respectively.

Figures (**c**) and (**d**) are the cross-correlation functions (6) when the received signals have been cross-correlated with base chirps of the **SF** equal to 10 and 9, respectively.

0.3 Path Loss Modeling

In order to obtain the maximum communication range of a LoRa signal at 2.4 GHz (further denoted as ***d***), we need to find the maximum link budget for which the signal can be received properly; i.e., at the receiver sensitivity ***PRX***.
This receiver sensitivity depends on two key factors: the used **spreading factor** (**SF**) and **bandwidth** (**BW**). While the **SF** can range from **5 to 12**, the possible bandwidths of **LoRa at 2.4 GHz** are:
203, **406**, **812** and **1625 kHz**.

Furthermore, the combination of a certain **SF** and **BW** results in a certain data rate, along with the receiver sensitivity, as shown in Table 2. The **raw data rate *Rb***, expressed in **kbit/s**, can be calculated as

(8)
$$R_b = \frac{SF * BW}{2^{SF}},$$

with **SF** and **BW** as defined in Table 1. As an example, a LoRa signal transmitted with an **SF** of **8** and a **BW** of **406 kHz** results in a receiver **sensitivity** of **–116 dBm** and a data rate of **12.69 kbit/s**.

The receiver sensitivities and data rates used in this work originate from the datasheet of the Semtech SX1280 LoRa module.

Table 2
Receiver sensitivities (***PRX*** in **dBm**) and corresponding data rates (***RD*** in) of the SX1280 LoRa module for every combination of spreading factors (**SFs**) and bandwidths (**Bws**).

	BW (kHz)							
	203		**406**		**812**		**1625**	
SF	P_{RX}	R_D	P_{RX}	R_D	P_{RX}	R_D	P_{RX}	R_D
5	−109	31.72	−107	63.44	−105	126.88	−99	253.91
6	−111	19.03	−110	38.06	−108	76.13	−103	152.34
7	−115	11.1	−113	22.2	−112	44.41	−106	88.87
8	−118	6.34	−116	12.69	−115	25.38	−109	50.78
9	−121	3.57	−119	7.14	−117	14.27	−111	28.56
10	−124	1.98	−122	3.96	−120	7.93	−114	15.87
11	−127	1.09	−125	2.18	−123	4.36	−117	8.73
12	−130	0.595	−128	1.19	−126	2.38	−120	4.76

The total link budget of a wireless communication signal propagating from the transmitter to receiver can be represented as:

(9)
$$P_{RX} = P_{TX} + G_{TX} - L_{TX} - L_p(d) + G_{RX} - L_{RX},$$

where ***PRX*** is the **received power** in **dBm**, ***PTX*** is the **transmission power** in **dBm**, ***GTX*** is the **antenna gain** at the **transmitter** in **dBi**, ***LTX*** is the **cable loss** in **dB**, ***Lp*(*d*)** is the **path loss** in **dB** in terms of the **function from the distance *d***, ***GRX*** is the **antenna gain** at the **receiver** in **dB** and ***LRX*** is the **cable loss** at the **receiver** in **dB**.

Except for the path loss, all these parameters are set to **typical values** which are commonly used when simulating a wireless communication link between two dipole antennas .
The values of these parameters are summarized in Table 1.

The path loss ***Lp*(*d*)** is defined as the **propagation loss** caused by the signal traveling from the transmitter to receiver over a distance ***d*.** The goal in this research is to **maximize *d*** while still being able to successfully receive the LoRa-modulated signal at the receiver.
For the sake of simplicity, the simulated LoRa signal contains **eight preamble symbols** and **no payload bytes**.
Depending on the environment, the path loss should be modeled differently. Therefore, in the next three subsections, we discuss **indoor**, **outdoor** (urban) and f**ree space** path loss models to translate the propagation loss into a distance between the transmitter and receiver.
The parameters required by these models are also summarized in Table 1. No fade margin is taken into account.

0.3.1 Free Space Environment

The first scenario can be described as a **free space environment** in which there is a **line of sight** (**LoS**, i.e., the primary Fresnel zone is to be at least 60% clear.) between the TX and RX locations.
In this case, we can use the widely used Free Space Path Loss (FPSL) model to evaluate the maximum communication range. This model calculates the loss between two isotropic radiators in free space, without considering any obstacles, reflections or interference.
The model solely relies on the **frequency** and **distance** between the transmitter and receiver to calculate the path loss:

(10)

$$L_{p,LoS}\left(d\right) = 32.44 + 20\log_{10}\left(f\right) + 20\log_{10}\left(d\right),$$

where ***f*** is in **MHz** and ***d*** is in **km**. The combination of a given **SF** and **BW** yields a certain sensitivity ***PRX***. Consequently, given the maximum path loss obtained from (9), we can calculate the maximum distance as :

(11)

$$d = 10^{(L_{p,LoS}(d) - 32.44 - 20\log_{10}(f))/20}.$$

0.3.2. Indoor Environment

In the second scenario, we evaluate the maximum communication range in an indoor environment. To this end, we slightly adapt a heuristic algorithm that was developed based on real measurements in an **office-like environment** . The path loss model is based on the Indoor **Dominant Path** (IDP) model, which focuses on the dominant path between the TX and RX location.
In general, the total path loss is the sum of the distance loss, accumulated **wall loss** and **interaction loss** and can be calculated as:

(12)

$$L_{p,in}\left(d\right) = L_{p_0}\left(d_0\right) + 10 * n * \log_{10}\left(\frac{d}{d_0}\right) + \sum_i L_{W_i} + \sum_j L_{B_j},$$

where ***Lp*0**(*d*0) represents the path loss at a distance ***d*0** and ***n*** is the **path loss exponent**. The accumulated wall loss is the sum of losses ***LWi*** caused by **each wall** along the dominant path. Finally, the interaction loss is the sum of losses ***LBj*** caused by all directional changes of the propagating signal.

Given the **semi-empirical** nature of this path loss model, some parameters need to be set to commonly used values in order to provide a generally applicable model that can predict ranges in other indoor environments.

Therefore, ***Lp*0**(*d*0) is set to **40 dB** at a distance ***d*0=1 m**. The path loss exponent is set to ***n*=5**, which is generally used for obstructed paths inside buildings. For the accumulated wall and interaction loss, values of 6 and 3 dB have been taken into account, as found specifically for the office-like environment. Consequently, the path loss model can be simplified to:

(13)

$$L_{p,in}(d) = 40 + 5 * 10 * \log_{10}(d) + 6 + 3.$$

Thus, the range can be empirically estimated based on the path loss:

(14)

$$d = 10^{(L_{p,in}(d)-49)/50}.$$

0.3.3. Urban Environment

An urban path loss model is used in the third scenario to evaluate the range of LoRa at 2.4 GHz in an **outdoor city-scale environment**. The Okumura–Hata Urban Path Loss model is an empirical model that is often used in sub-GHz wireless communication systems. While the COST-231 urban model extended its use up to 2 GHz, the Electronic Communication Committee (ECC) modified the original Okumura–Hata model to work with frequencies up to (and beyond) 3 GHz in the ECC-33 model. Therefore, the ECC-33 model is suitable to evaluate the maximum communication range of LoRa at 2.4 GHz in an urban environment.

The **path loss** equation for this model is given by:

(15)

$$L_{p,urban}(d) = A_{fs} + A_{bm} + G_b + G_m,$$

where ***Afs*** is the **free space attenuation**, ***Abm*** is the **basic median path loss**, ***Gb*** is the base station height gain factor and ***Gr*** is the receiver height gain factor, which can be calculated as:

(16 - 19)

$$A_{fs} = 92.4 + 20\log_{10}(d) + 20\log_{10}(f),$$

$$\begin{aligned} A_{bm} &= 20.41 + 9.83\log_{10}(d) + 7.894 log_{10}(f) \\ &\quad + 9.56[\log_{10}(f)]^2, \end{aligned}$$

$$G_b = \log_{10}\left(\frac{h_b}{200}\right)\left\{13.958 + 5.8[\log_{10}(d)]^2\right\}, \text{ and}$$

$$G_m = [42.57 + 13.7\log_{10} f]\,[\log_{10}(h_m) - 0.585]$$

for medium-sized urban environments. Given the complexity of this set of equations, we extract the maximum range by iterating over values of ***d* from 1 m to 10 km** and solving the optimization problem given a certain path loss ***Lp,urban*(*d*)**.

0.4 Range versus Data Rate: Results

Figure 2, Figure 3 and Figure 4 show the maximum communication range and corresponding data rate at each combination of SF and bandwidth for the free space and indoor and urban environments, respectively. In all cases, the highest possible data rate decreases in a logarithmic way when the communication range between the transmitter and receiver increases.

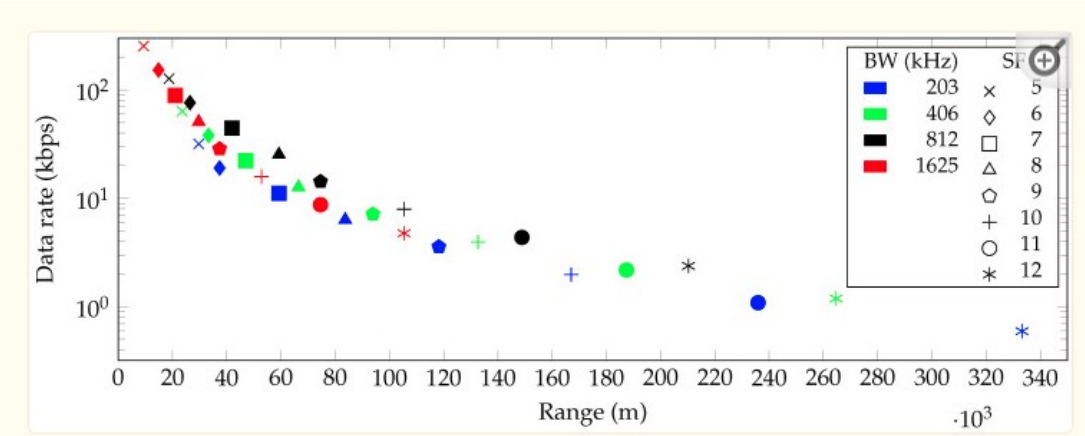


Fig 2. Communication range and data rate for every combination of spreading factor (SF) and bandwidth (BW) in a **free space line of sight** (**LoS**) environment.

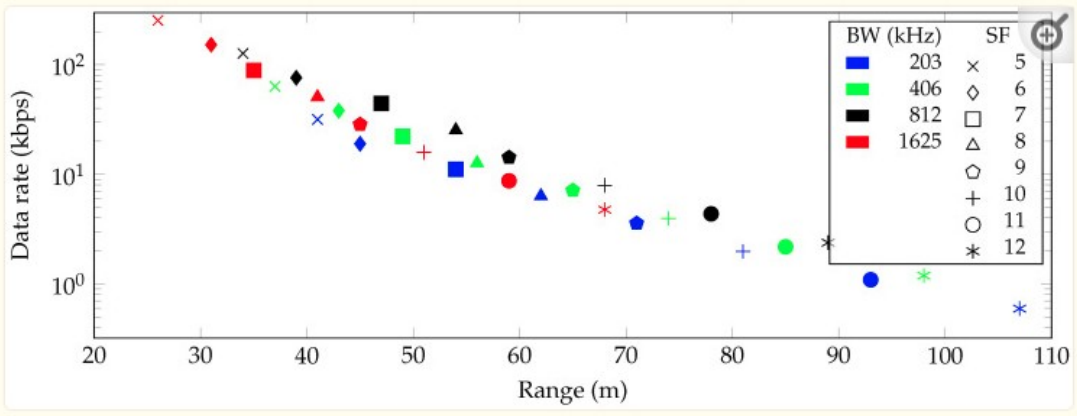


Fig 3. Communication range and data rate for every combination of spreading factor (SF) and bandwidth (BW) in an **indoor environment**.

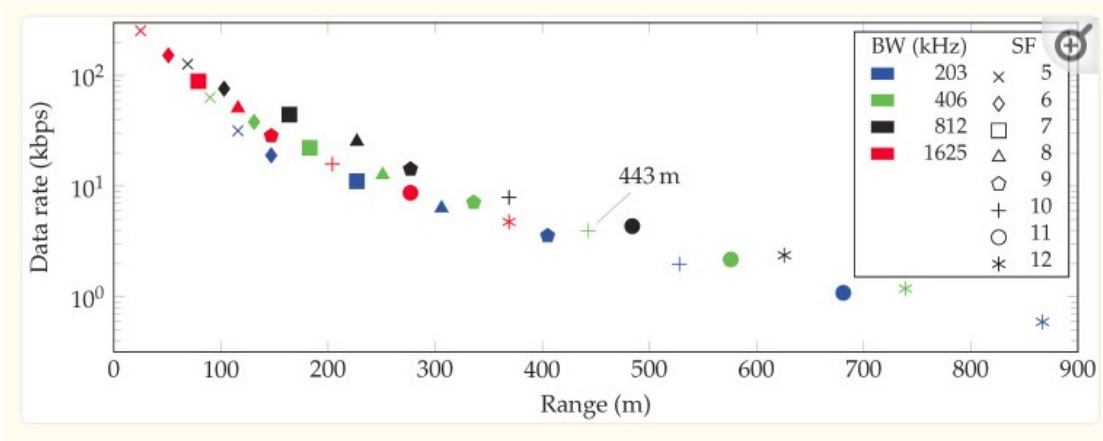


Fig 4. Communication range and data rate for every combination of spreading factor (SF) and bandwidth (BW) in an **urban environment**.

Using the Free Space Path Loss model, it is found that a 2.4 GHz LoRa signal can travel **up to 333 km** in free space and still be received properly. Obviously, this is only a theoretical range and cannot be realized in real-world environments.

The performance of a more realistic indoor path loss model has been visualized in Fig 3. When transmitting with a spreading factor of 12 and the lowest bandwidth (i.e., **203 kHz**), the path loss equals **150.5 dB**.

Consequently, a maximum communication range of **107 m** can be achieved. Furthermore, the highest possible data rate at that range becomes 0.595 kbit/s. At the other extreme, the highest achievable data rate of 253.91 kbit/s is possible at a range of up to **26 m**.

Finally, the communication range of the urban ECC-33 path loss model varies from **25 m** at the highest achievable data rate of **253.91 kbit/s** to **867 m** at the lowest possible data rate of **0.595 kbit/s**.

0.5 Discussion

We investigated the maximum communication range of LoRa in the 2.4 GHz band, which is defined as the maximum distance between a transmitter and receiver at which a LoRa-modulated message can be received properly.
Our investigations included three environments: LoS free space, NLoS indoor and urban outdoor. In all three scenarios, we maximized the range by reducing the bandwidth and increasing the spreading factor.

Based on the total link budget of the wireless communication system, including receiver sensitivity, antenna gains and cable losses, we were able to estimate the range of LoRa at 2.4 GHz. It is important to note that we did not include a fade margin in the link budget calculations. The fade margin can be defined as the level of received power in excess of that required for a specified minimum level of system performance.

The reason for excluding this loss parameter in (9) is the high variability of fade margin in different scenarios. For instance, a 5 dB fade margin decreases the maximum urban range from 867 m to 576 m, while a 10 dB fade margin further decreases the range to 369 m. Thus, this should be taken into account when analyzing the results. Nonetheless, the largest factor by far in a link budget is the path loss.

The free space line-of-sight scenario resulted in a theoretical maximum range of 333 km when transmitting at the highest SF and using the lowest bandwidth. In reality, the signal will always have to cope with obstacles, multipath propagation effects and interference with other signals.

Therefore, these ranges **will never be achieved in a real-world environment**. Nevertheless, the results of the Free Space Path Loss model are useful as a benchmark as they enable us to compare them with different frequencies and technologies. For instance, the maximum range of **LoRa at 868 MHz** calculated with the FSPL model equals **921 km**, **which is almost three times the range of LoRa at 2.4 GHz**.

For the indoor range estimation of 2.4 GHz LoRa, an indoor path loss model was evaluated. In order to provide the highest possible accuracy, a model based on real-world measurements was adopted and slightly modified, taking into account both wall and interaction loss. The estimated range varies from 26 m to 107 m, depending on the SF and BW. It should be noted that a path loss exponent of 5 was chosen, simulating an obstructed indoor environment. However, in an indoor LoS scenario, the range might therefore be increased.

Finally, the maximum communication range in an urban environment was found to be 867 m.
As indicated in Fig 4, the range at an SF of 10 and a BW equal to 406 kHz is 443 m. This is similar to the results of the experiments with the SX1280 chipset carried out by Wolf et al. [17]. They found that ToF ranging with the aforementioned SF and BW failed for ranges over about 500 m.
Although they only investigated the ranging feature at an SF of 10 and a BW equal to 406 kbit/s and 1625 kbit/s, this partially validates our range estimations of the ECC-33 path loss model.

Besides the communication range, we investigated the data rates for all combinations of SFs and BWs and consequently associated this information with the highest achievable range. The highest possible data rate of LoRa at 2.4 GHz equals 253.91 kbit/s, which is almost seven times higher than the maximum data rate of LoRa at 868 MHz. This data rate can be achieved if the distance between the transmitter and receiver is not greater than 9393 m, 26 m and 25 m in a free space, indoor and urban environment, respectively.

Some significant differences in terms of range arise when comparing LoRa to other technologies operating in the 2.4 GHz band. As mentioned earlier, the range of the latest Bluetooth standard equals 50 m and 165 m in an indoor and outdoor environment, respectively. Moreover, the maximum range of 2.4 GHz Wi-Fi networks typically varies around 100 m. Thus, the outdoor range of LoRa is more than five times larger than the outdoor range of BLE 5 and more than eight times larger compared to typical IEEE 802.11 networks.

This is mainly due to the lower bandwidth and data rates used in LoRa, as well as the robustness of the LoRa-modulated signal. These numbers clearly indicate the significant difference in intended applications between LoRa (such as long-range communication and localization) and Wi-Fi and Bluetooth (such as video and audio streaming).

Since LoRa modulation at 2.4 GHz has a higher bandwidth than LoRa modulation at 868 MHz, **the rising edge of a signal pulse can be determined more accurately**.

Therefore, we expect that time-based localization methods for this technology will result in lower estimation errors. However, the results in this paper show that it is not possible to achieve the same long communication ranges as LoRa at 868 MHz and with other sub-GHz LPWANs.

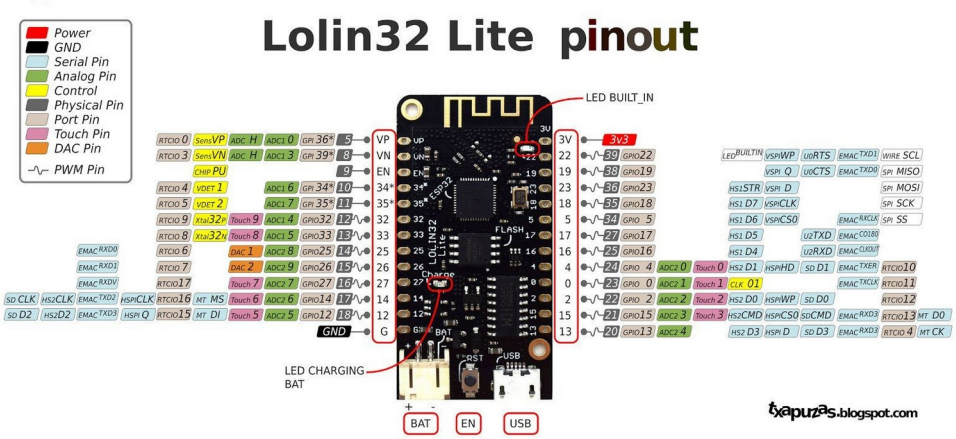
Therefore, **more LoRa receivers have to be deployed to cover wide areas**, which makes it a less feasible solution to build large public networks. On the other hand, 2.4 GHz LoRa is an interesting option for both communication and localization in privately deployed networks that are purposed for asset tracking and monitoring in large warehouses, construction sites, farms, etc.

Lab 1

Data transmission with LoRa 2.4GHz on ESP32 (Lolin D32) board and DevKit

1.1 The hardware

Our lab is based on IoT DevKit with ESP32 Lolin32 D32 and SX1280 modem.



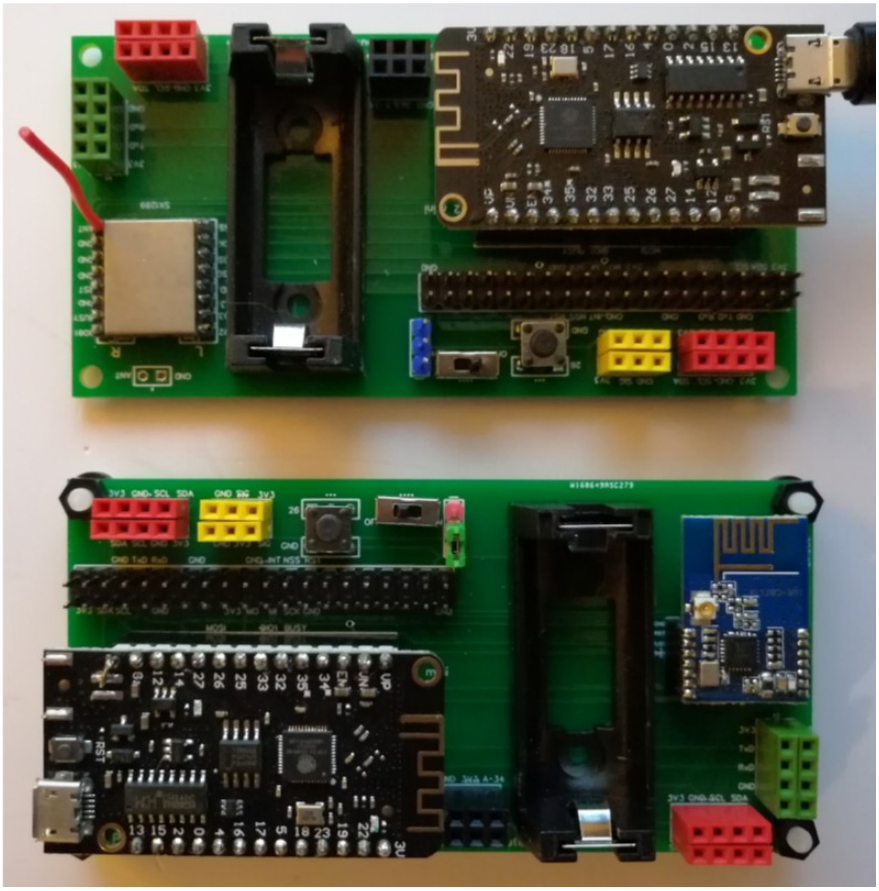


Fig 1.1 ESP32 (Lolin32 Lite) pinout and **Pomme-Pi ONE LoRa 2.4 GHz** boards with SX1280 modem

1.2 The software

The programming of the board is done in C/C++ language with Arduino IDE – Tools
You need **`RadioLib`** library from here:

`https://github.com/jgromes/RadioLib`

Files to take into account and adapt to our board(s).

`utilities.h`, **`boards.h`**

In the **`utilities.h`** file we define the pinout of our board. This is necessary to activate correctly the connections to LoRa modem (SX1280) (bus **SPI**) , to the on-board OLED screen (bus **I2C**), and SD card reader (secondary bus **SPI**).

In the **`boards.h`** file we define the functions/methods to activate and use the LoRa modem, OLED screen, and the integrated SD card reader.

In our case we incorporate directly the selected elements of these files into our code.

1.3 Sending and receiving LoRa data frames with SX1280 modem

In this section we study simple send and receive operations with LoRa (2.4 GHz) modem. We are using **`RadioLib.h`** library to control the modem.

1.3.1 Transmitter code

RadioLib SX128x Transmit code with Interrupts:

This example transmits LoRa packets with **one second delays** between them. Each packet contains **up to 256 bytes of data**, in the form of:

- **Arduino String**
- **null-terminated char array (C-string)**
- **arbitrary binary data (byte array)**

Other modules from SX128x family can also be used.

For default module settings, see the **wiki page**:

https://github.com/jgromes/RadioLib/wiki/Default-configuration#sx128x---lora-modem

For full API reference, see the **GitHub Pages**:

`https://jgromes.github.io/RadioLib/`

1.3.1.1 Complete code of transmitter without interruption :

```
#include <RadioLib.h>
#include <Wire.h>

// modem connections: SPI +

#define RADIO_SCLK_PIN              5
#define RADIO_MISO_PIN              19
#define RADIO_MOSI_PIN              27
#define RADIO_CS_PIN                18
#define RADIO_DI0_PIN               26
#define RADIO_RST_PIN               23
#define RADIO_DIO1_PIN              33
#define RADIO_BUSY_PIN              32

#define I2C_SDA              12
#define I2C_SCL              14

SX1280 radio = new Module(RADIO_CS_PIN, RADIO_DI0_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN);

int transmissionState = 0;
volatile bool receivedFlag = false;
volatile bool enableInterrupt = true;

uint32_t counter = 0;

void setFlag(void)
{
    // check if the interrupt is enabled
    if (!enableInterrupt) {
        return;
    }
   // we got a packet, set the flag
    receivedFlag = true;
}

void setup() {
  Serial.begin(115200);
  Serial.println("initBoard");
  SPI.begin(RADIO_SCLK_PIN, RADIO_MISO_PIN, RADIO_MOSI_PIN);
  Wire.begin(I2C_SDA, I2C_SCL);
  delay(1500);
  Serial.print("[SX1280] Initializing ... ");
```

```
  delay(1500);
  int state = radio.begin();
      if (state == 0) {
        Serial.println("success!");
    } else {
        Serial.print("failed, code ");
        Serial.println(state);
        while (true);
    }
radio.setDio1Action(setFlag);
}

void loop()
{

  Serial.println(millis());
  Serial.printf("[SX1280] Sending another packet :%d\n ", counter);
  transmissionState = radio.startTransmit((uint8_t *)&counter, 4);
  Serial.println(transmissionState);
  counter++;
  delay(1000);

}
```

1.3.1.2 Transmitter code with interruption signal on `RADIO_DIO1_PIN (33)`

```
#include <RadioLib.h>
#include <Wire.h>
// modem connections: SPI +
#define RADIO_SCLK_PIN              5
#define RADIO_MISO_PIN              19
#define RADIO_MOSI_PIN              27
#define RADIO_CS_PIN                18
#define RADIO_DIO_PIN               26
#define RADIO_RST_PIN               23
#define RADIO_DIO1_PIN              33
#define RADIO_BUSY_PIN              32
#define I2C_SDA              12
#define I2C_SCL              14

SX1280 radio = new Module(RADIO_CS_PIN, RADIO_DIO1_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN);
int transmissionState = 0;
volatile bool transmittedFlag = false;
volatile bool enableInterrupt = true;

uint32_t counter = 0;

void setFlag(void)
{
    // check if the interrupt is enabled
    if (!enableInterrupt) {
        return;
    }
   // we got a packet, set the flag
    transmittedFlag = true;
}

void setup() {
  Serial.begin(115200);
  Serial.println("initBoard");
  SPI.begin(RADIO_SCLK_PIN, RADIO_MISO_PIN, RADIO_MOSI_PIN);
  Wire.begin(I2C_SDA, I2C_SCL);
  delay(1500);
  Serial.print("[SX1280] Initializing ... ");
  delay(1500);
  int state = radio.begin();
      if (state == 0) {
        Serial.println("success!");
    } else {
        Serial.print("failed, code ");
        Serial.println(state);
        while (true);
    }
radio.setDio1Action(setFlag);
transmissionState = radio.startTransmit((uint8_t *)&counter, 4);
}
```

```
void loop()
{
if (transmittedFlag) {
        // disable the interrupt service routine while
        // processing the data
        enableInterrupt = false;
        // reset flag
        transmittedFlag = false;
        if (transmissionState == RADIOLIB_ERR_NONE) {
            // packet was successfully sent
            Serial.println(F("transmission finished!"));

            // NOTE: when using interrupt-driven transmit method,
            //       it is not possible to automatically measure
            //       transmission data rate using getDataRate()

        } else {
            Serial.print(F("failed, code "));
            Serial.println(transmissionState);
        }

  Serial.println(millis());
  delay(1000);
  Serial.printf("[SX1280] Sending another packet :%d\n ", counter);
  transmissionState = radio.startTransmit((uint8_t *)&counter, 4);
  Serial.println(transmissionState);
  counter++;
  enableInterrupt = true;
  }
}
```

1.3.3 Receiver code

1.3.3.1 RadioLib SX128x Receive with Interrupts

This example **listens for LoRa transmissions** and tries to receive them. Once a packet is received, an interrupt is triggered. To successfully receive data, the following settings have to be the same on both transmitter and receiver:

- **carrier frequency**
- **bandwidth**
- **spreading factor**
- **coding rate**
- **sync word**

Other modules from SX128x family can also be used.

For default module settings, see the **wiki page**:

```
https://github.com/jgromes/RadioLib/wiki/Default-configuration#sx128x---lora-modem
```

For full API reference, see the **GitHub Pages:**

https://jgromes.github.io/RadioLib/

Complete code

```
#include <RadioLib.h>
#include <Wire.h>

// modem connections: SPI +

#define RADIO_SCLK_PIN              5
#define RADIO_MISO_PIN              19
#define RADIO_MOSI_PIN              27
#define RADIO_CS_PIN                18
#define RADIO_DI0_PIN               26
#define RADIO_RST_PIN               23
```

```
#define RADIO_DIO1_PIN              33
#define RADIO_BUSY_PIN              32

#define I2C_SDA              12
#define I2C_SCL              14

SX1280 radio = new Module(RADIO_CS_PIN, RADIO_DIO1_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN);

int transmissionState = 0;
volatile bool receivedFlag = false;
volatile bool enableInterrupt = true;

uint32_t counter = 0;

void setFlag(void)
{
    // check if the interrupt is enabled
    if (!enableInterrupt) {
        return;
    }
    // we sent a packet, set the flag
    receivedFlag = true;
}

void setup() {
  Serial.begin(115200);
  Serial.println("initBoard");
  SPI.begin(RADIO_SCLK_PIN, RADIO_MISO_PIN, RADIO_MOSI_PIN);
  Wire.begin(I2C_SDA, I2C_SCL);
  delay(1500);
  Serial.print("[SX1280] Initializing ... ");
  delay(1500);
  int state = radio.begin();
      if (state == 0) {
        Serial.println("success!");
    } else {
        Serial.print("failed, code ");
        Serial.println(state);
        while (true);
    }
 radio.setDio1Action(setFlag);
 Serial.print("[SX1280] Starting to listen ... ");
    state = radio.startReceive();
    if (state == 0) {
        Serial.println("success!");
    } else {
        Serial.print("failed, code ");
        Serial.println(state);
        while (true);
    }
    enableInterrupt=true;
}

void loop()
{
  if (receivedFlag)
  {
        // disable the interrupt service routine while
        enableInterrupt = false;
        // reset flag
        receivedFlag = false;
        Serial.println("[SX1280] Receiving packet");
        counter=0;
        int state = radio.readData((uint8_t *)&counter, 4);
        Serial.println(state);
        Serial.println(radio.getRSSI());
        Serial.println(counter);
        radio.startReceive();
        enableInterrupt = true;
  }
  delay(100);
}
```

Lab 2

LoRa 2.4 GHz LILYGO T-BEAM – communication

2.1 The hardware

Our lab is based on the following MCU/LoRa board and our main base board including the battery and a set of IO interfaces for sensors and actuators.

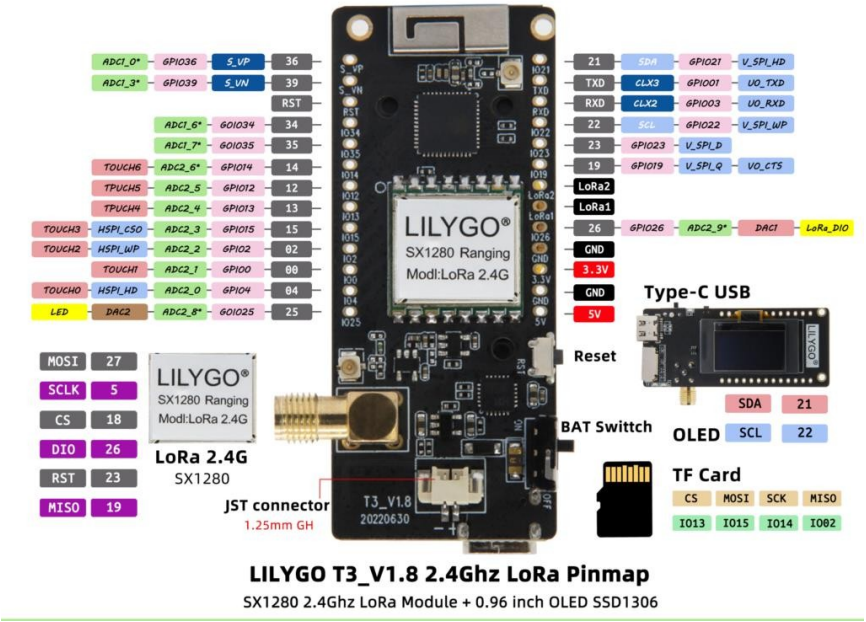


Fig 2.1 `LILYGO_T3_V1_8` board with SX1280 modem

This board is incorporated into our main board:

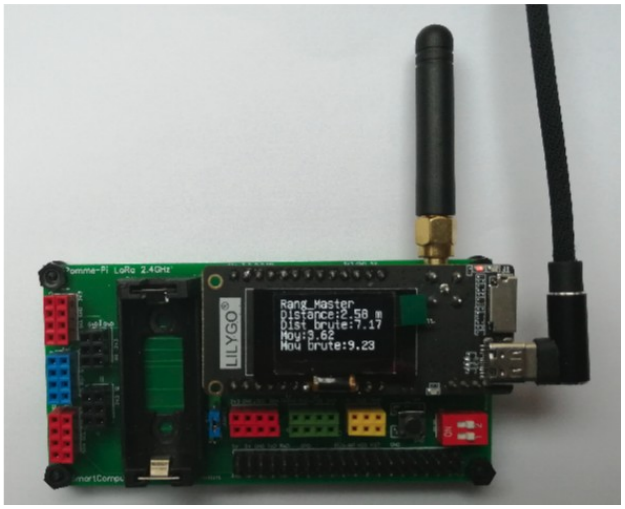


Fig 2.2 Pomme-Pi LoRa 2.4 GHz base board (Master node)

2.2 The software

The programming of the board is done in C/C++ language with Arduino IDE – Tools

```
// This is our board

#define LILYGO_T3_V1_8
```

Type of board **T-Beam**

port: **/dev/ttyACM0**

You need **RadioLib** library from here:

https://github.com/jgromes/RadioLib

File to to include:

utilities.h, **boards.h**

In the **utilities.h** file we define the pinout of our board. This is necessary to activate correctly the connections to LoRa modem (SX1280) (bus **SPI**) , to the on-board OLED screen (bus **I2C**), and SD card reader (secondary bus **SPI**).

In the **boards.h** file we define the functions/methods to activate and use the LoRa modem, OLED screen, and the integrated SD card reader.

```
// utilities.h

#define LILYGO_T3_V1_8

#define UNUSE_PIN                   (0)

#define I2C_SDA                     21
#define I2C_SCL                     22
#define OLED_RST                    UNUSE_PIN

#define RADIO_SCLK_PIN              5
#define RADIO_MISO_PIN              19
#define RADIO_MOSI_PIN              27
#define RADIO_CS_PIN                18
#define RADIO_DI0_PIN               26
#define RADIO_RST_PIN               23
#define RADIO_DIO1_PIN              33
#define RADIO_BUSY_PIN              32

#define SDCARD_MOSI                 15
#define SDCARD_MISO                 2
#define SDCARD_SCLK                 14
#define SDCARD_CS                   13

#define BOARD_LED                   25
#define LED_ON                      HIGH

#define ADC_PIN                     35

#define HAS_SDCARD
#define HAS_DISPLAY
```

```
// boards.h

#include <Arduino.h>
#include <SPI.h>
#include <Wire.h>
#include <Ticker.h>
#include "utilities.h"
```

```
#ifdef HAS_SDCARD
#include <SD.h>
#include <FS.h>
#endif

#ifdef HAS_DISPLAY
#include <U8g2lib.h>
U8G2_SSD1306_128X64_NONAME_F_HW_I2C *u8g2 = nullptr;
#endif

Ticker ledTicker;
#if defined(LILYGO_TBeam_V1_0) || defined(LILYGO_TBeam_V1_1)
#include <axp20x.h>
AXP20X_Class PMU;

bool initPMU()
{
    if (PMU.begin(Wire, AXP192_SLAVE_ADDRESS) == AXP_FAIL) {
        return false;
    }
    /*
     * The charging indicator can be turned on or off
     * * * */
    // PMU.setChgLEDMode(LED_BLINK_4HZ);
    /*
    * The default ESP32 power supply has been turned on,
    * no need to set, please do not set it, if it is turned off,
    * it will not be able to program
    *
    *   PMU.setDCDC3Voltage(3300);
    *   PMU.setPowerOutPut(AXP192_DCDC3, AXP202_ON);
    *
    * * * */
    /*
     *   Turn off unused power sources to save power
     * **/
    PMU.setPowerOutPut(AXP192_DCDC1, AXP202_OFF);
    PMU.setPowerOutPut(AXP192_DCDC2, AXP202_OFF);
    PMU.setPowerOutPut(AXP192_LDO2, AXP202_OFF);
    PMU.setPowerOutPut(AXP192_LDO3, AXP202_OFF);
    PMU.setPowerOutPut(AXP192_EXTEN, AXP202_OFF);
    /*
     * Set the power of LoRa and GPS module to 3.3V
     **/
    PMU.setLDO2Voltage(3300);   //LoRa VDD
    PMU.setLDO3Voltage(3300);   //GPS  VDD
    PMU.setDCDC1Voltage(3300);  //3.3V Pin next to 21 and 22 is controlled by DCDC1
    PMU.setPowerOutPut(AXP192_DCDC1, AXP202_ON);
    PMU.setPowerOutPut(AXP192_LDO2, AXP202_ON);
    PMU.setPowerOutPut(AXP192_LDO3, AXP202_ON);
    pinMode(PMU_IRQ, INPUT_PULLUP);
    attachInterrupt(PMU_IRQ, [] {
        // pmu_irq = true;
    }, FALLING);
    PMU.adc1Enable(AXP202_VBUS_VOL_ADC1 |
                   AXP202_VBUS_CUR_ADC1 |
                   AXP202_BATT_CUR_ADC1 |
                   AXP202_BATT_VOL_ADC1,
                   AXP202_ON);
    PMU.enableIRQ(AXP202_VBUS_REMOVED_IRQ |
                  AXP202_VBUS_CONNECT_IRQ |
                  AXP202_BATT_REMOVED_IRQ |
                  AXP202_BATT_CONNECT_IRQ,
                  AXP202_ON);
    PMU.clearIRQ();
    return true;
}

void disablePeripherals()
{
    PMU.setPowerOutPut(AXP192_DCDC1, AXP202_OFF);
    PMU.setPowerOutPut(AXP192_LDO2, AXP202_OFF);
    PMU.setPowerOutPut(AXP192_LDO3, AXP202_OFF);
}
#else
#define initPMU()
```

```
#define disablePeripherals()
#endif

SPIClass SDSPI(HSPI);



void initBoard()
{
    Serial.begin(115200);
    Serial.println("initBoard");
    SPI.begin(RADIO_SCLK_PIN, RADIO_MISO_PIN, RADIO_MOSI_PIN);
    Wire.begin(I2C_SDA, I2C_SCL);
#ifdef HAS_GPS
    Serial1.begin(GPS_BAUD_RATE, SERIAL_8N1, GPS_RX_PIN, GPS_TX_PIN);
#endif
#if OLED_RST
    pinMode(OLED_RST, OUTPUT);
    digitalWrite(OLED_RST, HIGH); delay(20);
    digitalWrite(OLED_RST, LOW);  delay(20);
    digitalWrite(OLED_RST, HIGH); delay(20);
#endif
    initPMU();
#ifdef BOARD_LED
    /*
    * T-BeamV1.0, V1.1 LED defaults to low level as trun on,
    * so it needs to be forced to pull up
    * * * * */
#if LED_ON == LOW
    gpio_hold_dis(GPIO_NUM_4);
#endif
    pinMode(BOARD_LED, OUTPUT);
    ledTicker.attach_ms(500, []() {
        static bool level;
        digitalWrite(BOARD_LED, level);
        level = !level;
    });
#endif
#ifdef HAS_DISPLAY
    Wire.beginTransmission(0x3C);
    if (Wire.endTransmission() == 0) {
        Serial.println("Started OLED");
        u8g2 = new U8G2_SSD1306_128X64_NONAME_F_HW_I2C(U8G2_R0, U8X8_PIN_NONE);
        u8g2->begin();
        u8g2->clearBuffer();
        u8g2->setFlipMode(0);
        u8g2->setFontMode(1); // Transparent
        u8g2->setDrawColor(1);
        u8g2->setFontDirection(0);
        u8g2->firstPage();
        do {
            u8g2->setFont(u8g2_font_inb19_mr);
            u8g2->drawStr(0, 30, "LilyGo");
            u8g2->drawHLine(2, 35, 47);
            u8g2->drawHLine(3, 36, 47);
            u8g2->drawVLine(45, 32, 12);
            u8g2->drawVLine(46, 33, 12);
            u8g2->setFont(u8g2_font_inb19_mf);
            u8g2->drawStr(58, 60, "LoRa");
        } while ( u8g2->nextPage() );
        u8g2->sendBuffer();
        u8g2->setFont(u8g2_font_fur11_tf);
        delay(3000);
    }
#endif
#ifdef HAS_SDCARD
    if (u8g2) {
        u8g2->setFont(u8g2_font_ncenB08_tr);
    }
    pinMode(SDCARD_MISO, INPUT_PULLUP);
    SDSPI.begin(SDCARD_SCLK, SDCARD_MISO, SDCARD_MOSI, SDCARD_CS);
    if (u8g2) {
        u8g2->clearBuffer();
    }
    if (!SD.begin(SDCARD_CS, SDSPI)) {
        Serial.println("setupSDCard FAIL");
```

```
        if (u8g2) {
            do {
                u8g2->setCursor(0, 16);
                u8g2->println( "SDCard FAILED");;
            } while ( u8g2->nextPage() );
        }
    } else {
        uint32_t cardSize = SD.cardSize() / (1024 * 1024);
        if (u8g2) {
            do {
                u8g2->setCursor(0, 16);
                u8g2->print( "SDCard:");;
                u8g2->print(cardSize / 1024.0);;
                u8g2->println(" GB");;
            } while ( u8g2->nextPage() );
        }
        Serial.print("setupSDCard PASS . SIZE = ");
        Serial.print(cardSize / 1024.0);
        Serial.println(" GB");
    }
    if (u8g2) {
        u8g2->sendBuffer();
    }
    delay(3000);
#endif

}
```

2.3 Sending and receiving LoRa data frames with SX1280 modem

In this section we study simple send and receive operations with LoRa (2.4 GHz) modem. We are using **`RadioLib.h`** library to control the modem.

2.3.1 Transmitter code

RadioLib SX128x Transmit code with Interrupts:

This example transmits LoRa packets with **one second delays** between them. Each packet contains **up to 256 bytes of data**, in the form of:

- **Arduino String**
- **null-terminated char array (C-string)**
- **arbitrary binary data (byte array)**

Other modules from SX128x family can also be used.

For default module settings, see the **wiki page**:

https://github.com/jgromes/RadioLib/wiki/Default-configuration#sx128x---lora-modem

For full API reference, see the **GitHub Pages**:

`https://jgromes.github.io/RadioLib/`

Complete code:

```
#include <RadioLib.h>
#include "include/boards.h"
#define ERR_NONE 0
SX1280 radio = new Module(RADIO_CS_PIN, RADIO_DI0_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN);
// save transmission state between loops
int transmissionState = ERR_NONE;
// flag to indicate that a packet was sent
volatile bool transmittedFlag = false;
// disable interrupt when it's not needed
volatile bool enableInterrupt = true;
uint32_t counter = 0;
// this function is called when a complete packet
// is transmitted by the module
// IMPORTANT: this function MUST be 'void' type
//            and MUST NOT have any arguments!
void setFlag(void)
{
    // check if the interrupt is enabled
    if (!enableInterrupt) {
        return;
    }
    // we sent a packet, set the flag
    transmittedFlag = true;
}

void setup()
{
    initBoard();
    Serial.begin(9600);
    // When the power is turned on, a delay is required.
    delay(1500);
    // initialize SX1280 with default settings
    Serial.print(F("[SX1280] Initializing ... "));
    int state = radio.begin();
#ifdef HAS_DISPLAY
    if (u8g2) {
        if (state != ERR_NONE) {
            u8g2->clearBuffer();
            u8g2->drawStr(0, 12, "Initializing: FAIL!");
            u8g2->sendBuffer();
        }
    }
#endif
```

```
    if (state == ERR_NONE) {
        Serial.println(F("success!"));
    } else {
        Serial.print(F("failed, code "));
        Serial.println(state);
        while (true);
    }
    // set the function that will be called
    // when packet transmission is finished
    radio.setDio1Action(setFlag);
    // start transmitting the first packet
    Serial.print(F("[SX1280] Sending first packet ... "));
    // you can transmit C-string or Arduino string up to
    // 256 characters long
    // transmissionState = radio.startTransmit("Hello World!");
    // you can also transmit byte array up to 256 bytes long
    // byte byteArr[] = {0x01, 0x23, 0x45, 0x67,
    //                   0x89, 0xAB, 0xCD, 0xEF
    //                  };
    // state = radio.startTransmit(byteArr, 8);
    transmissionState = radio.startTransmit((uint8_t *)&counter, 4);
}

void loop()
{
// check if the previous transmission finished
    if (transmittedFlag) {
        // disable the interrupt service routine while
        // processing the data
        enableInterrupt = false;
        // reset flag
        transmittedFlag = false;
        if (transmissionState == ERR_NONE) {
            // packet was successfully sent
            Serial.println(F("transmission finished!"));
            // NOTE: when using interrupt-driven transmit method,
            //       it is not possible to automatically measure
            //       transmission data rate using getDataRate()
#ifdef HAS_DISPLAY
            if (u8g2) {
                u8g2->clearBuffer();
                u8g2->drawStr(0, 12, "Transmitting: OK!");
                u8g2->drawStr(0, 30, ("TX:" + String(counter)).c_str());
                u8g2->sendBuffer();
            }
#endif
        } else {
            Serial.print(F("failed, code "));
            Serial.println(transmissionState);
        }
        // wait a second before transmitting again
        delay(500);
        // send another one
        Serial.print(F("[SX1280] Sending another packet ... "));
        // you can transmit C-string or Arduino string up to
        // 256 characters long
        // transmissionState = radio.startTransmit("Hello World!");
        // you can also transmit byte array up to 256 bytes long
        /*
          byte byteArr[] = {0x01, 0x23, 0x45, 0x67,
                            0x89, 0xAB, 0xCD, 0xEF};
          int state = radio.startTransmit(byteArr, 8);
        */
        transmissionState = radio.startTransmit((uint8_t *)&counter, 4);
        counter++;
        // we're ready to send more packets,
        // enable interrupt service routine
        enableInterrupt = true;
    }
}
```

2.3.2 Receiver code

RadioLib SX128x Receive with Interrupts:
This example **listens for LoRa transmissions** and tries to receive them. Once a packet is received, an interrupt is triggered. To successfully receive data, the following settings have to be the same on both transmitter and receiver:

- **carrier frequency**
- **bandwidth**
- **spreading factor**
- **coding rate**
- **sync word**

Other modules from SX128x family can also be used.

For default module settings, see the **wiki page**:

https://github.com/jgromes/RadioLib/wiki/Default-configuration#sx128x---lora-modem

For full API reference, see the **GitHub Pages:**

https://jgromes.github.io/RadioLib/

Complete code

```
#include <RadioLib.h>
#include "boards.h"
#define ERR_NONE 0
#define ERR_CRC_MISMATCH 1
SX1280 radio = new Module(RADIO_CS_PIN, RADIO_DIO1_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN);
// flag to indicate that a packet was received
volatile bool receivedFlag = false;
// disable interrupt when it's not needed
volatile bool enableInterrupt = true;
// this function is called when a complete packet
// is received by the module
// IMPORTANT: this function MUST be 'void' type
//            and MUST NOT have any arguments!
void setFlag(void)
{
    // check if the interrupt is enabled
    if (!enableInterrupt) {
        return;
            }
    // we got a packet, set the flag
    receivedFlag = true;
}

void setup()
{
    initBoard();
    Serial.begin(9600);
    // When the power is turned on, a delay is required.
    delay(1500);
    // initialize SX1280 with default settings
    Serial.print(F("[SX1280] Initializing ... "));
    int state = radio.begin();
    if (u8g2) {
        if (state != ERR_NONE) {
            u8g2->clearBuffer();
            u8g2->drawStr(0, 12, "Initializing: FAIL!");
            u8g2->sendBuffer();
        }
    }
    if (state == ERR_NONE) {
        Serial.println(F("success!"));
    } else {
        Serial.print(F("failed, code "));
        Serial.println(state);
        while (true);
    }
```

```
    // set the function that will be called
    // when packet transmission is finished
    radio.setDio1Action(setFlag);
    // start listening for LoRa packets
    Serial.print(F("[SX1280] Starting to listen ... "));
    state = radio.startReceive();
    if (state == ERR_NONE) {
        Serial.println(F("success!"));
    } else {
        Serial.print(F("failed, code "));
        Serial.println(state);
        while (true);
    }
    // if needed, 'listen' mode can be disabled by calling
    // any of the following methods:
    //
    // radio.standby()
    // radio.sleep()
    // radio.transmit();
    // radio.receive();
    // radio.readData();
    // radio.scanChannel();
}

void loop()
{LILYGO_T3_V1_8
    // check if the flag is set
    if (receivedFlag) {
        // disable the interrupt service routine while
        // processing the data
        enableInterrupt = false;
        // reset flag
        receivedFlag = false;
        // you can read received data as an Arduino String
        // String str;
        // int state = radio.readData(str);
        uint32_t counter;
        int state = radio.readData((uint8_t *)&counter, 4);
               // you can also read received data as byte array
        /*
          byte byteArr[8];
          int state = radio.readData(byteArr, 8);
        */
        if (state == ERR_NONE) {
            // packet was successfully received
            Serial.println(F("[SX1280] Received packet!"));
            // print data of the packet
            Serial.print(F("[SX1280] Data:\t\t"));
            Serial.println(counter);
            // print RSSI (Received Signal Strength Indicator)
            Serial.print(F("[SX1280] RSSI:\t\t"));
            Serial.print(radio.getRSSI());
            Serial.println(F(" dBm"));
            // print SNR (Signal-to-Noise Ratio)
            Serial.print(F("[SX1280] SNR:\t\t"));
            Serial.print(radio.getSNR());
            Serial.println(F(" dB"));
#ifdef HAS_DISPLAY
            if (u8g2) {
                u8g2->clearBuffer();
                char buf[256];
                u8g2->drawStr(0, 12, "Received OK!");
                snprintf(buf, sizeof(buf), "RX:%u", counter);
                u8g2->drawStr(0, 26, buf);
                snprintf(buf, sizeof(buf), "RSSI:%.2f", radio.getRSSI());
                u8g2->drawStr(0, 40, buf);
                snprintf(buf, sizeof(buf), "SNR:%.2f", radio.getSNR());
                u8g2->drawStr(0, 54, buf);
                u8g2->sendBuffer();
            }
#endif
        } else if (state == ERR_CRC_MISMATCH) {
            // packet was received, but is malformed
            Serial.println(F("[SX1280] CRC error!"));

        } else {
```

```
            // some other error occurred
            Serial.print(F("[SX1280] Failed, code "));
            Serial.println(state);
        }
        // put module back to listen mode
        radio.startReceive();
        // we're ready to receive more packets,
        // enable interrupt service routine
        enableInterrupt = true;
    }
}
```

Lab 3

Long Distance Ranging with LoRa 2.4 GHz

3.1 Introduction

All practical radio communication employs a protocol, this is the set of rules determining when a radio has to be ready to receive messages destined for it or transmit its own messages to another radio.

Ranging operations are similar because the ranging transactions need to be **coordinated**, but when and how these operations are performed also affects the resulting ranging accuracy. In addition to this, the collected results need conversion into a physical range. Depending upon how we have performed the ranging, we may have additional metadata such as signal strength or frequency error that we can also use to augment the precision of the ranging measurement.

This introduction describes both the communications protocol and post processing performed by the SX1280 modem and Pomme-Pi LoRa 2.4 GHz kit when in 'Outdoor Ranging Mode'.

In addition to a brief overview of the basic principles of ranging, we walk you through both the protocol and the post processing of the ranging results, also explaining the rationale behind each step and alternative approaches.

3.2 Ranging Operation

The ranging distance measurement is based upon the round trip time of flight (**RTToF**) of a signal between a pair of radios. The basic principle is illustrated below. Here, one radio assumes the role of ranging **Requester** and another the role of ranging **Responder**.

The ranging **Requester** sends a ranging request to the **Responder**, which sends a **synchronised response** back to the **Requester**. The Requester measures and interpolates the time elapsed between the ranging request, **TRequest**, and response **TResponse**.

This **measured time** reported by the Requester is hence the **round-trip time** between the **Requester** and **Responder**. With all propagation occurring at the speed of light, the measured time is equivalent to the measured round-trip distance - with some **additional timing errors** (ε)

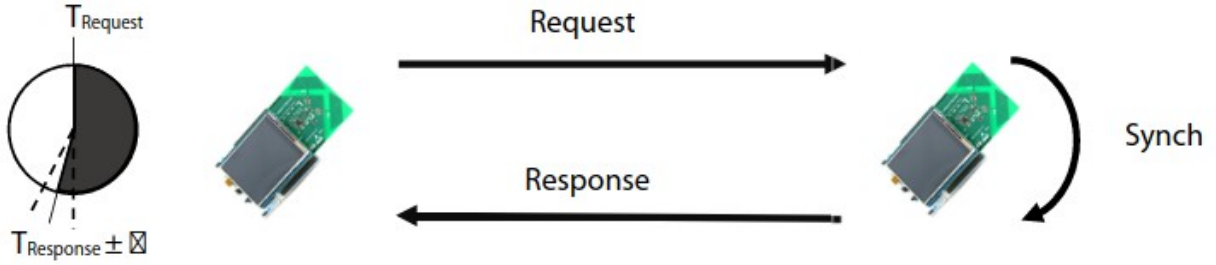


Fig. 3.1 Principle of **RTToF** ranging

When considering the post processing of the **RTToF distance measurement,** we have to consider these sources of error and, where possible, correct them. This in turn has an effect on the way we collect the distance measurement results themselves. We therefore begin by examining these sources of error and how to mitigate them.

3.2.1 Radio Ranging Errors

Before we begin, we need to draw an important delineation between **static sources of error** and those that can **change** during the operation of the ranging measurement system. It is relatively **simple to correct static bias**, here we want to consider the tougher case of dynamic, time varying, errors. So, to simplify our discussion, **we assume that the static sources of measurement error been corrected by calibration**.

The remaining **dynamic sources** of error can be further divided into those i**nternal to the radio** and those due to the **channel** in which the ranging signals propagate.

Assuming a perfect communication channel, we see only errors due to circuit level phenomenon, namely:

• Reference **oscillator drift** and
• Analog **group delay**.

Before we look at the influence of the communication channel and propagation on the ranging accuracy, we examine these two sources of error and their correction.

3.2.1.1 Reference Oscillator Error

The **Requester's** timing measurement and the **Responder's synchronisation** are performed using the local crystal reference **oscillator of the SX1280**. Thus, **any offset** in timing between the **Requester** and **Responder** crystal oscillators will result in a distance measurement error.

Correcting Reference Oscillator Error
Because we use the same reference oscillator to derive both the **timer for ranging** operations and the **RF carrier frequency** for the 2.4 GHz transmission, we can use the measure of **frequency error** between transmitter and receiver to indicate reliably and accurately the timing (so distance) offset between **Requester** and **Responder**.

The figure below shows the distance measurement correction that should be applied in response to the given frequency (so timing) error.

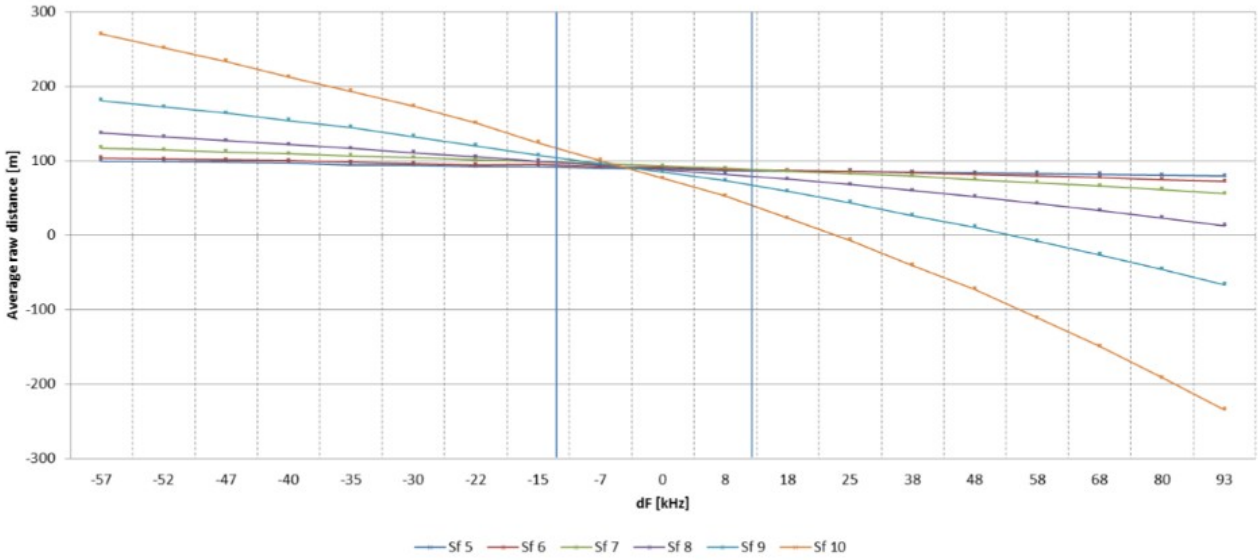


Fig. 3.2 Distance Error versus **Frequency Offset** between **Requester** and **Responder**

3.2.1.2 Analogue Group Delay Error

The SX1280 employs an **Automatic Gain Control** (AGC) to adapt the **LNA gain** of the **receiver** to the received signal strength. This process is standard to almost all radio receivers to allow the reception of both low power signals at the limits of sensitivity and high power signals at short range

Setting	Gain [dB]
13	Max
12	Max -2
11	Max -4
10	Max -6
9	Max -8
8	Max -12
7	Max -18
6	Max -24
5	Max -30
4	Max -36
3	Max -42
2	Max -48
1	Max -54

Fig. 3.3 The SX1280 **LNA Gain** (Outlined) and **Range of Gain Settings**

The impact of this on the operation on **RTToF** measurement is that the **delay through the LNA** changes as a **function of the amplifier gain**.
We have measured this in the figure below, with the measurement setup also shown in the inset image.
Each point corresponds to a raw ranging measurement between the **Requester** and **Responder** over a 125m (electrical length, 100 m physical length) cable. Here we can see that as we vary the attenuation between **Requester** and **Responder**, at a fixed distance, from high input power (low attenuation) on the left to low input power (high attenuation) on the right - we experience over **8 m of measurement error.**

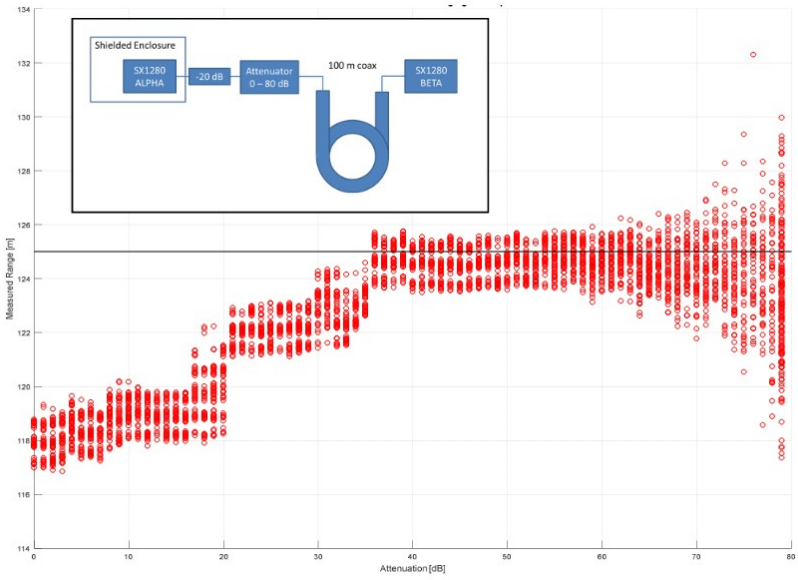


Fig. 3.4 Measured **Requester**-**Responder** Distance, at a **Fixed Distance**, as a **Function of Attenuation.**

Correcting Analogue Group Delay Error
Correction of the variable delay due to the LNA gain setting would be possible by **sharing information between** the ranging **Requester** and **Responder** about which gain level was used during each ranging exchange.

This approach would impose the following problems:

1) A **real time processing operation to monitor gain levels** of both the **Requester** and **Responder** during each exchange. (i.e. Polling the internal radio registers to measure the gain used for each exchange).
2) The requirement for additional **communication overhead** to send the LNA gain information from **Responder** to **Requester**. (i.e. Send the gain used by the **Responder** back to the **Requester**).

In order to avoid this **we make use of the RSSI** of the ranging exchange measured by the ranging **Requester**. In doing so we are assuming that the channel remains static for the duration of a single ranging exchange and, secondly, that the signal power seen by the **Requester** gives an indication of the receiver gain used by both **Requester** and **Responder**.

The **Ranging RSSI** differs slightly from the RSSI measured conventionally, instead of indicating the power in absolute power (**dBm**) the **ranging RSSI** indicates the received signal power relative to a **signal power threshold** in dB.

Correction based upon the **ranging RSSI** is straightforward. Based upon measurements made in a coaxial cable and variable attenuation, we construct a look-up-table of **ranging RSSI versus distance measurement** correction. The **LUT** contains 160 entries per **SF-BW** combination so, for clarity, the **LUT** data for **SF9 1600 kHz** is plotted below.

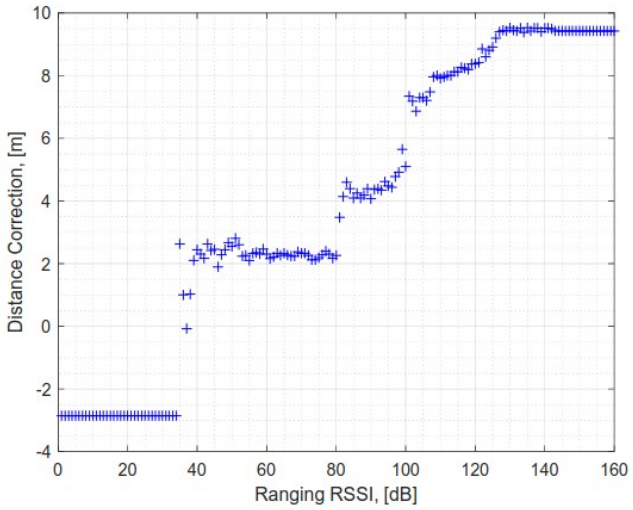


Fig. 3.5 Applied **distance correction** as a function of **Ranging RSSI for SF9 BW** = **1600 kHz** which is stored as a **LUT** in firmware

3.2.2 Channel Effects

With the ranging errors inherent to the radio system addressed, we must also consider the **influence of the communication channel** itself on the ranging result.

3.2.2.1 Channel Selectivity & Multipath

At **80 MH**z wide, the **2.4 GHz ISM band** is much wider than the highest SX1280 LoRa bandwidth of **1.6 MHz**. Consequently, we can see significant variation in the ranging performance across the 2.4 GHz ISM band. Below is an extreme case of the point in question. In this example **Requester** and **Responder** are 150 m apart with obstructions between them (**non-Line-of-Sight**, **nLoS**, channel conditions)

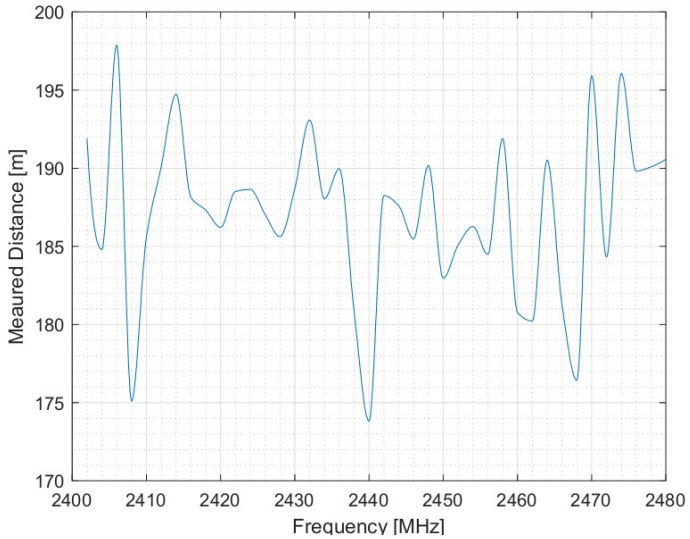


Fig 3.6. nLoS Ranging Measurement Results at **150 m**

This phenomenon arises because the wave propagating between the **Requester** and **Responder** happens on **different paths**, each path being subject to **different delays**. In the frequency domain, in the case shown above, this delay dispersion translates into **frequency selectivity** on the scale of the 2.4 GHz ISM band.

In addition to frequency selectivity, the other influence of **nLoS** propagation between **Requester** and **Responder** is the additional bias (extra distance) that the measurement result will exhibit.

In the measurement above, we see an additional 30 to 40 m compared with the **ground truth distance**.

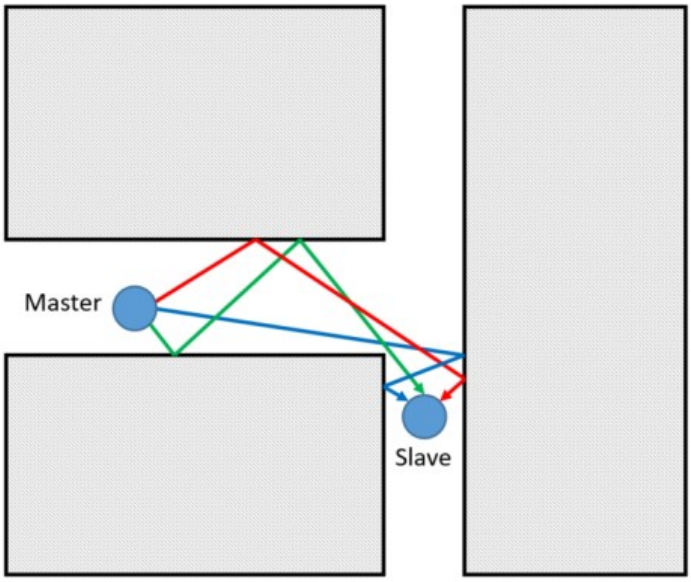


Fig 3.7. Illustration of **Multipath Propagation** between a Pair of **non-Line of Sight** (**nLoS**) Ranging Radios

The key to understand the origin of both effects is shown in the plan view (above) of the **Requester** transmitting the ranging request to the **Responder**. (Noting that the same will apply for the return, ranging response, propagation).
In this scenario, we have **nLoS propagation reflected** by some surrounding scattering buildings. The red green and blue lines denote the **reflected paths**. In this **nLoS** layout we see that there are multiple reflected paths from **Requester** to **Responder**, each with differing path lengths. We also see that the **reflected nLoS paths are all longer** than the physical distance from the **Requester** to **Responder**.

This is what causes:

1) The **over estimation** of the measure distance between radios, as the overall reflected distance is higher.
2) The frequency selectivity of the response of the band due to the delay dispersion of the multiple reflected paths.

3.2.2.2 Compensating Frequency Selectivity

Whilst there is no easy way to mitigate the range over estimation effect of multipath, the delay spread, so selectivity effects caused by multipath propagation can be lessened by the use of **frequency and antenna diversity**. Of the two diversity schemes, frequency diversity is the most important.

Frequency Diversity
As can be seen from the measured 2.4 GHz ISM band response, some frequencies will give less accurate range results than others. The solution to this is frequency diversity, i.e. **performing many ranging exchanges** across the band to help reduce the influence of measurement on a less accurate channel.

Antenna Diversity
An alternative to changing the frequency at which we perform the measurement, to change the delay spread seen by the radio, is to change antenna. The three types of antenna diversity are **spatial**, **polarisation** and **pattern diversity**. However, what is important to note here is that by changing antenna we aim to change the multipath environment between **Requester** and **Responder** so further diminishing the influence of inaccurate channels.

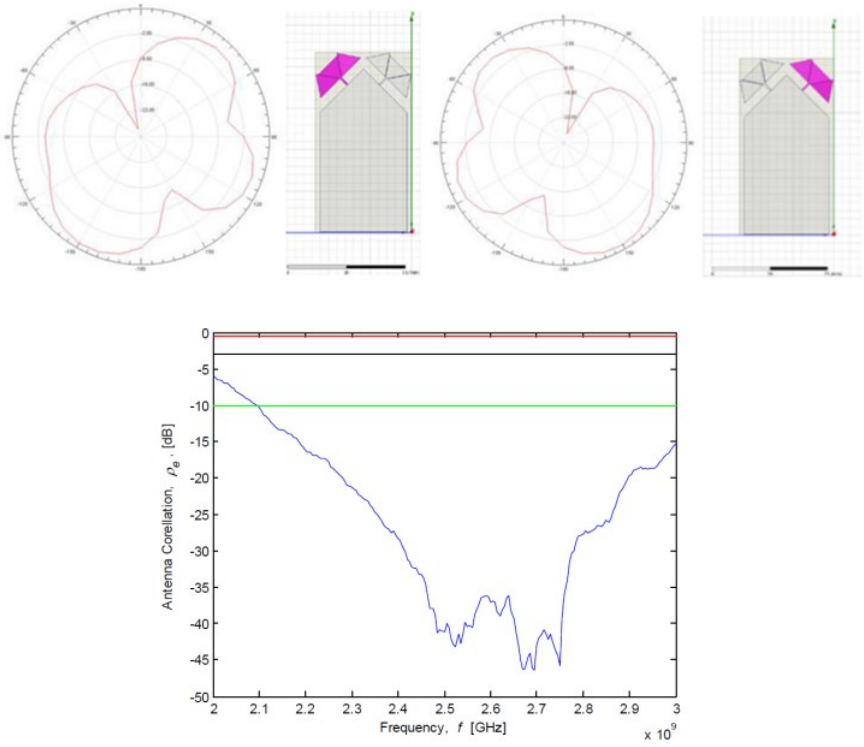


Fig 3.8. The Antenna Diversity for SX1280 . (Top) the radiation patterns of each antenna and (Bottom) the Measured Antenna Envelope Correlation Coefficient.

Antenna diversity is sometimes beyond the space or cost constraints of an application.
Nonetheless, the example of the SX1280 development kit is shown above. Here the top image shows the pattern diversity of the two antennas wherein the antenna radiation patterns are designed to differ from each other.

3.2.2.3 Correcting Measurement Bias by Fingerprinting

Fingerprinting is the technical term used to describe the process of **fitting measured ranging** or localisation **estimates**, to a specific environment. In our case, it is used to remove the bias introduced by multipath.

Whilst a detailed discussion of this subject is beyond the scope of this presentation, we examine the line-of-sight case, noting that these techniques could be modified for ranging measurements in other environments.

Why do we need to correct for the line of sight case? Because even the line-of-sight channel is subject to multipath. The results from a line of sight measurement between a pair of evaluation kits is shown below.

Here we see the measured result versus **Ground Truth** (GT). Even in line of sight conditions we see some range underestimation at short range, some overestimation between **10 to 80 m** and then underestimation beyond this. Thus, even in the line of sight case, we perform a polynomial curve fit (red line) with a view to using the polynomial as a correction.

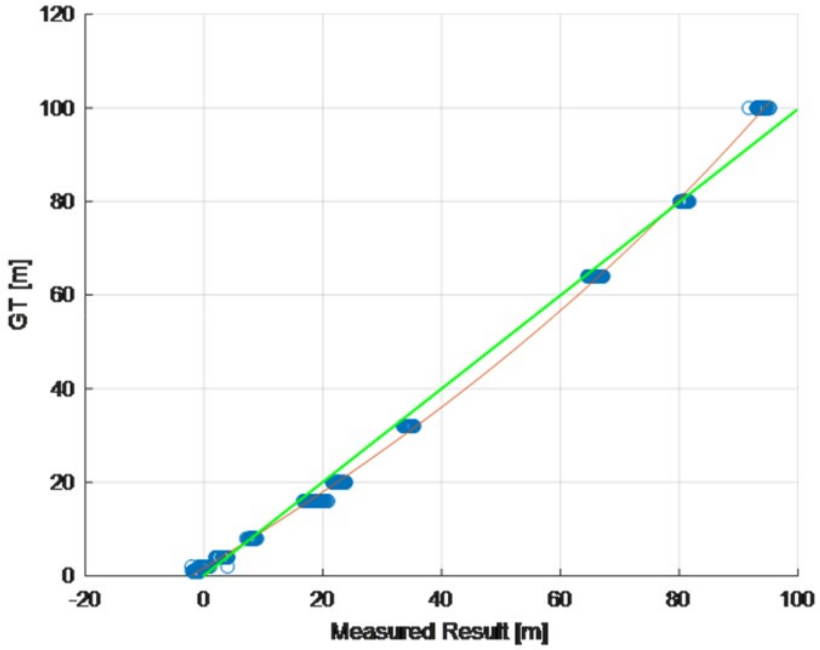


Fig 3.9. **Curve Correction**: Measurement Results versus Ground Truth (blue) and Polynomial Fit (red)

3.3 Implementation: A Practical RTToF Measurement System

3.3.1. Introduction

In this part, we take the abstract approaches outlined in the previous Section and apply them in practice. This application takes two forms:

1. The ranging protocol used to recover the ranging distance measurements and
2. The post processing of the set of results recovered and its conversion into a measured distance.

3.3.2. The Ranging Protocol

The role of the ranging protocol is **twofold**: to ensure that the radios are in the **right configuration at the right time** and to **gather adequate ranging results**. In addition to this we recover the associated metadata, based upon our understanding from the previous section, to further improve our measurement accuracy.

The basic protocol we use between the **Requester** and **Responder** during a ranging measurement is illustrated below.

The ranging exchange begins with a **communication phase**. In a real application this would allow the **Responder** radio to remain in a **duty cycled sleep mode** to reduce power consumption.

In our demonstration ranging application the **Responder** remains **in receive mode** until it received the first LoRa communication. Once we have established connectivity between the **Requester** and **Responder** we then start a series of **frequency hopped ranging exchanges**, each on a **mutually defined channel**.

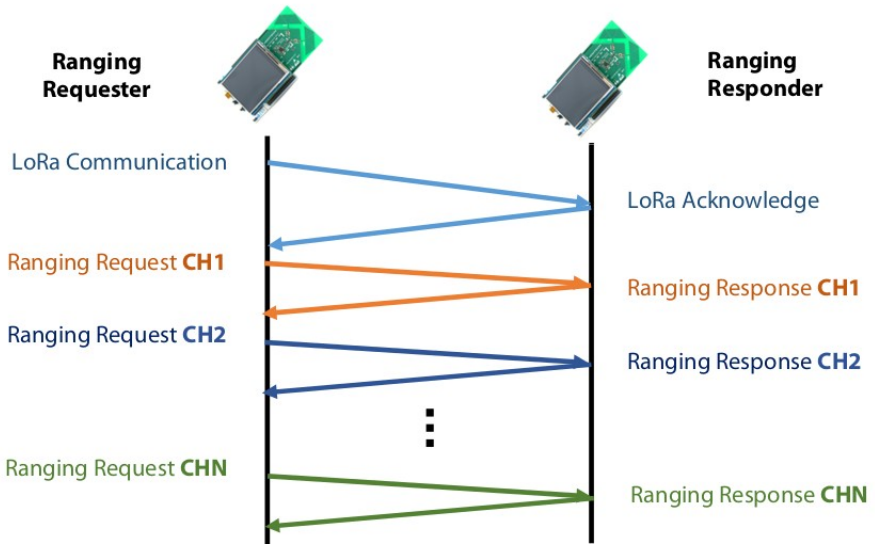


Fig 3.10. The **Ranging Protocol**

3.3.3 The Channel Plan

The **frequency hopped exchanges** use the **BLE channel plan**. In our example, we perform a single exchange on a single frequency and **then hop to the next frequency** for the following exchange.
Due to the frequency selectivity of the ISM band hopping across multiple channels allows us the best chance of avoiding inaccurate channels.

In the implementation on our kit the number of hopping channels is variable. If we define fewer than 40 ranging exchanges per result, say 30, then we only proceed up to the 30th channel in the hopping sequence.

Where we define more than 40 exchanges per result, say 60, then the first 40 exchanges will hop over the full 40 channels, then the first 20 channels in the list will be replayed. The frequency hopping sequence used in our ranging example. The red highlighted channels denote the BLE advertising channels.

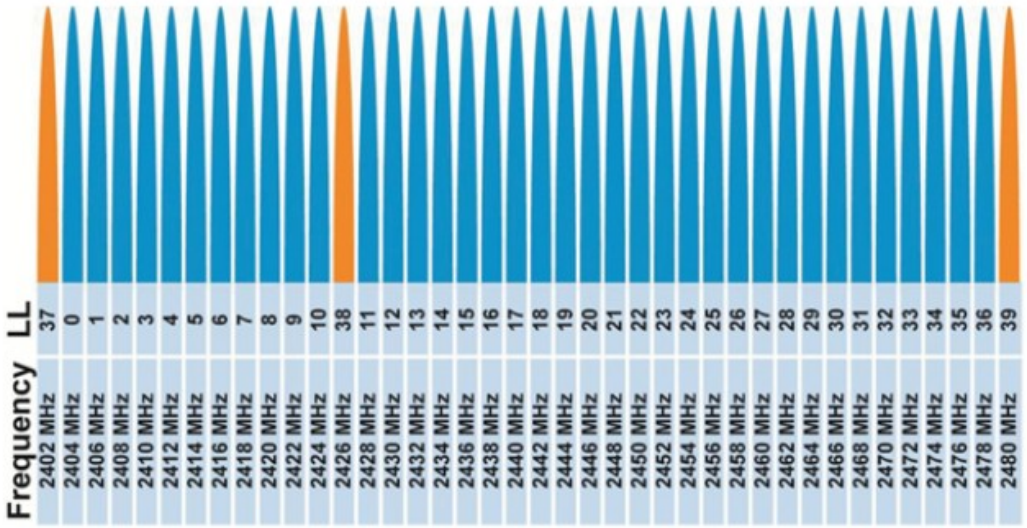


Fig 3.11. Frequency Hopping Scheme

3.3.4 Raw Results

The delineation between communication and ranging exchanges is important. This is because there are different metadata available that can help us improve the ranging accuracy available from each exchange.

Communication Phase
Given there is only one communication exchange per ranging measurement, we only have one of the following measurements per ranging result.

> **FEI The Frequency Error Indicator**: Indicates the frequency error measured by the **Requester** upon receiving the communication packet from the **Responder**.
>
> **Communication RSSI :** The received signal strength of the communication packet.

Ranging Phase
Each ranging result is based upon multiple ranging exchanges, we have one of the quantities below per hopping channel, so multiple results per ranging measurement sequence.

> **Ranging Measurement**: The ranging measurement itself is a 2's compliment hexadecimal number proportional to the round trip time of flight.
> **Ranging RSSI:** The signal power seen by the **Requester** during reception of the ranging exchange.

3.3.5 Processing the Ranging Results

With these results collated from both the communication and ranging phases, we can perform the range calculation and correction process. Here we outline the full post processing flow as performed in the evaluation kit.
Here we also examine snippets of source code, to see the relevant code, please consult the SX1280 MBED site.

Step 1: Conversion from ranging round trip time of flight to distance in m (multiplication by ranging LSB and divide by 2). The raw result converted into metres is given in the line

```
val = ( double )complement2( valLsb, 24 ) / ( double ) LoRaBandwidth( ) * 36621.09375;
```

Where the value **36621.09375 = (c / (2^12)) / 2** comes from the formula documented in the datasheet.
`valLsb` is the ranging result in 2's compliment hexadecimal.

This conversion is applied to each **RTToF** result recovered from the SX1280 and stored in an array of **`RawRngResult[]`.**

Step 2: Apply a correction based upon the FEI measurement from the communication exchange. This assumes that the frequency error is a linear curve with certain gradient. We stock the gradients as a function of **Sf** and **BW** as shown below:

```
const double RNG_FGRAD_0400[] = { −0.148, −0.214, −0.419, −0.853, −1.686, −3.423 };
const doubleRNG_FGRAD_0800[] = { −0.041, −0.811, −0.218, −0.429, −0.853, −1.737 };
const double RNG_FGRAD_1600[] = { 0.103, −0.041, −0.101, −0.211, −0.424, −0.87 };
```

For example for **SF9**, **1600 kHz** the **RngFeiFactor = -0.424**. the corrected result is given by subtracting the calculated error as shown below:

```
RawRngResults[i] = RawRngResults[i] − ( RngFeiFactor * RngFei / 1000 );
```

Where RawRngResult[i] is the ith result in the array of raw results and RngFei is the FEI value in Hz read from the first LoRa communication packet.

Step 3: The **LNA Compensation** of **SF9 1600 kHz** is as shown below. Here the **RSSI Ranging** is used directly as the look-up index for the distance correction. This correction is applied to each raw result:

```
// Generated LUT and Polynomial Values
#include "rangingCorrection_defines.h"
const double RangingCorrectionSF9BW1600[NUMBER_OF_FACTORS_PER_SFBW] =
{
−2.8555,
−2.8555,
−2.8555,
−2.8555,
−2.8555,
−2.8555,
−2.
```

Step 4: Take the **median value** of the remaining filtered results. The code below is used to determine the median, taking care to also cover the case where the two center most values must be averaged (if there are an even number of ranging exchanges from which to derive the median).

```
if ((RngResultIndex % 2) == 0)
{
median = (RawRngResults[RngResultIndex/2] + RawRngResults[(RngResultIndex/2) − 1])/2.0;
}
else
{
median = RawRngResults[RngResultIndex/2];
}
```

From this point we no longer deal with an array of values, but the **single filtered output** from the median. Filtered at least in the sense that the median has been found empirically to give much better immunity to outliers that easily distort the average. This is very important in the case where we are seeking immunity from inaccurate channels.

Step 5: Apply a polynomial correction to linearize our results in the line-of-sight case. The polynomial coefficients are stored as shown below:

```
const RangingCorrectionPolynomes_t correctionRangingPolynomeSF9BW1600 = {
.order = 7,
.coefficients = {
6.2781e−10,
−2.6256e−07,
3.7632e−05,
−0.0022408,
0.05252,
0.47936,
0.46183,
}
};
```

These are derived from real field measurements. For guidance on collecting data in the field please see our ranging measurement app note. Based upon field measurement we use curve fitting to derive the correction polynomial.

Because each **SF** and **BW** can have a different polynomial, here the correction is applied by simply looping through and accumulating the corrected polynomial – this allows us to accommodate different order polynomials for each **SF** and **BW** combination.

It is also important to note that the polynomial correction is only valid over the range of charachterized distances, here in the range **up to 100 m**.

```
const RangingCorrectionPolynomes_t *polynome = RangingCorrectionPolynomesPerSfBw[sf_index
][bw_index];
double correctedValue = 0.0;
double correctionCoeff = 0;
for(uint8_t order = 0; order < polynome->order; order++){
correctionCoeff = polynome->coefficients[order] * pow(median, polynome
>order - order - 1);
correctedValue += correctionCoeff;
}
return correctedValue;
```

Complete the corrected value returned by this function is the final output of the corrected range for outdoor ranging measurements in our **LoS** scenario.

3.3.6. Before and After

The plot below shows the difference that the result processing correction makes to the absolute ranging accuracy. These results were measured at a fixed SF and bandwidth of SF9 and 1600 kHz respectively

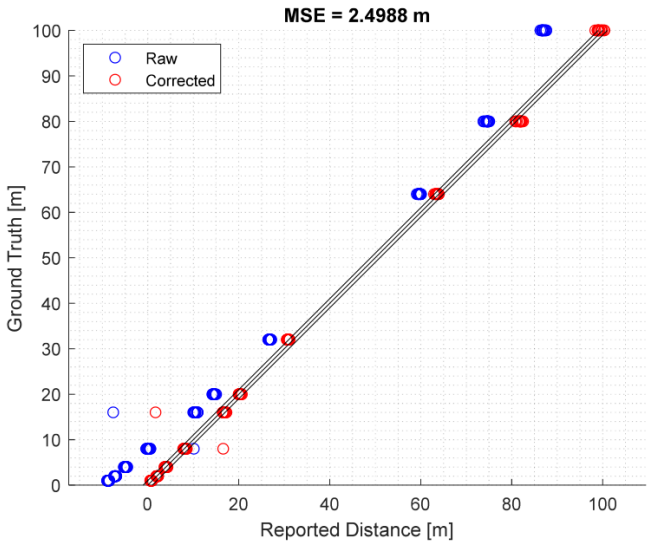


Fig 3.12. Before and After, the **Ranging Accuracy in LoS** using the techniques described here

The **blue points** show the result of simply taking the median ranging result over 40 frequency hopped exchanges in a **LoS** environment. The black line shows the ideal **x=y scale** that we should see between measured and **Ground Truth distance**.

With the full correction process applied as outlined above we obtain the results shown in red. It is important to note that the data set plotted here is not the training data with which the polynomial curve correction was generated but a separate cross validation data set.

In **conclusion**, it is possible to **drastically reduce the bias of ranging results and increase the accuracy by correcting for known and measurable errors** in the form of LNA gain and frequency error. Finally, typical environmental influences can be corrected through the use of statistical filtering and polynomial curve fitting to increase absolute accuracy.

3.4 Programming with simple configuration: Slave/Master

The following examples show the **Master**/**Slave** configuration (codes) for ranging application.

3.4.1 Ranging Master

You need **SX128XLT.h** library that may be found here:

https://github.com/StuartsProjects/SX12XX-LoRa/tree/master/examples/SX128x_examples/Ranging

```
#include "Arduino.h"
#include <SPI.h>
#include <SX128XLT.h>
#include "boards.h"

#define LORA_DEVICE DEVICE_SX1280                //we need to define the device we are using


//*******  Setup LoRa Parameters Here ! ***************
const uint32_t Frequency = 2445000000;           //frequency of transmissions in hz
const int32_t Offset = 0;                        //offset frequency in hz for calibration purposes
const uint8_t Bandwidth = LORA_BW_0800;          //LoRa bandwidth
const uint8_t SpreadingFactor = LORA_SF8;        //LoRa spreading factor
const uint8_t CodeRate = LORA_CR_4_5;            //LoRa coding rate
const uint16_t Calibration = 11350;              //Manual Ranging calibrarion value
const int8_t RangingTXPower = 12;                //Transmit power used
const uint32_t RangingAddress = 16;              //must match address in recever
const uint16_t  waittimemS = 10000;              //wait this long in mS for packet before assuming
timeout
const uint16_t  TXtimeoutmS = 5000;              //ranging TX timeout in mS
const uint16_t  packet_delaymS = 0;              //forced extra delay in mS between ranging requests
const uint16_t  rangeingcount = 10;               //number of times ranging is cqarried out for each
distance measurment
float distance_adjustment = 1.0000;              //adjustment factor to calculated distance

#define ENABLEOLED                               //enable this define to use display
#define ENABLEDISPLAY                            //enable this define to use display

SX128XLT LT;

uint16_t rangeing_errors, rangeings_valid, rangeing_results;
uint16_t IrqStatus;
uint32_t endwaitmS, range_result_sum, range_result_average;
float distance, distance_sum, distance_average;
bool ranging_error;
int32_t range_result;
int16_t RangingRSSI;

void led_Flash(uint16_t flashes, uint16_t delaymS);

void setup()
{
    pinMode(BOARD_LED, OUTPUT);   //setup pin as output for indicator LED
    led_Flash(4, 125);     //two quick LED flashes to indicate program start
    initBoard();
    Serial.begin(9600);
    delay(100);
    Serial.println(F("Ranging Master Starting"));
    SPI.begin(RADIO_SCLK_PIN, RADIO_MISO_PIN, RADIO_MOSI_PIN);

    if (LT.begin(RADIO_CS_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN, RADIO_DIO1_PIN, LORA_DEVICE)) {
        Serial.println(F("Device found"));
        led_Flash(2, 125);
        delay(1000);
    } else {
        Serial.println(F("No device responding"));
        u8g2->clearBuffer();
        u8g2->drawStr(0, 12, "No device responding");
        u8g2->sendBuffer();
        while (1) {
            led_Flash(50, 50);                                  //long fast speed flash indicates
device error
        }
    }
```

```
    LT.setupRanging(Frequency, Offset, SpreadingFactor, Bandwidth, CodeRate, RangingAddress,
RANGING_MASTER);
    LT.setRangingCalibration(Calibration);
//override automatic lookup of calibration value from library table
    Serial.println();
    LT.printModemSettings();        //reads and prints the configured LoRa settings, useful check
    Serial.println();
    LT.printOperatingSettings();
    //reads and prints the configured operating settings, useful check
    Serial.println();
    Serial.println();
    LT.printRegisters(0x900, 0x9FF);
    //print contents of device registers, normally 0x900 to 0x9FF
    Serial.println();
    Serial.println();

#ifdef ENABLEDISPLAY
    Serial.println("Display Enabled");
    u8g2->setFont(u8g2_font_unifont_t_chinese2);  // use chinese2 for all the glyphs of "你好世界"
    u8g2->setFontDirection(0);
    char buf[256];
    u8g2->clearBuffer();
    u8g2->drawStr(0, 12, "Ranging RAW Ready");
    snprintf(buf, sizeof(buf), "Power:%.d dBm", RangingTXPower);
    u8g2->drawStr(0, 12 * 2, buf);
    snprintf(buf, sizeof(buf), "Cal: %d ", Calibration);
    u8g2->drawStr(0, 12 * 3, buf);
    snprintf(buf, sizeof(buf), "Adjust: %d ", distance_adjustment);
    u8g2->sendBuffer();
#endif

    Serial.print(F("Address "));
    Serial.println(RangingAddress);
    Serial.print(F("CalibrationValue "));
    Serial.println(LT.getSetCalibrationValue());
    Serial.println(F("Ranging master RAW ready"));
    delay(2000);
}

void loop()
{
    uint8_t index;
    distance_sum = 0;
    range_result_sum = 0;
    rangeing_results = 0;                   //count of valid results in each loop
    for (index = 1; index <= rangeingcount; index++) {
        Serial.println(F("Start Ranging"));
        LT.transmitRanging(RangingAddress, TXtimeoutmS, RangingTXPower, WAIT_TX);
        IrqStatus = LT.readIrqStatus();
        if (IrqStatus & IRQ_RANGING_MASTER_RESULT_VALID) {
            rangeing_results++;
            rangeings_valid++;
            digitalWrite(BOARD_LED, HIGH);
            Serial.print(F("Valid"));
            range_result = LT.getRangingResultRegValue(RANGING_RESULT_RAW);
            Serial.print(F(",Register,"));
            Serial.print(range_result);
            if (range_result > 800000) {
                range_result = 0;
            }
            range_result_sum = range_result_sum + range_result;
            distance = LT.getRangingDistance(RANGING_RESULT_RAW, range_result, distance_adjustment);
            distance_sum = distance_sum + distance;
            Serial.print(F(",Distance,"));
            Serial.print(distance, 1);
            RangingRSSI = LT.getRangingRSSI();
            digitalWrite(BOARD_LED, LOW);
        } else {
            rangeing_errors++;
            distance = 0;
            range_result = 0;
            Serial.print(F("NotValid"));
            Serial.print(F(",Irq,"));
            Serial.print(IrqStatus, HEX);
        }
```

```
        delay(packet_delaymS);
        if (index == rangeingcount) {
            range_result_average = (range_result_sum / rangeing_results);
            if (rangeing_results == 0) {
                distance_average = 0;
            } else {
                distance_average = (distance_sum / rangeing_results);
            }
            Serial.print(F(",TotalValid,"));
            Serial.print(rangeings_valid);
            Serial.print(F(",TotalErrors,"));
            Serial.print(rangeing_errors);
            Serial.print(F(",AverageRAWResult,"));
            Serial.print(range_result_average);
            Serial.print(F(",AverageDistance,"));
            Serial.print(distance_average, 1);

#ifdef ENABLEDISPLAY
            u8g2->clearBuffer();
            char buf[256];
            u8g2->drawStr(0, 12, "Rang_Master");
            snprintf(buf, sizeof(buf), "Distance:%.2f m", distance_average);
            u8g2->drawStr(0, 12 * 2, buf);
            snprintf(buf, sizeof(buf), "RSSI: %d dBm", RangingRSSI);
            u8g2->drawStr(0, 12 * 3, buf);
            // snprintf(buf, sizeof(buf), "OK:  %d ", rangeings_valid);
            // u8g2->drawStr(0, 12 * 4, buf);
            // snprintf(buf, sizeof(buf), "Err:  %d ", rangeing_errors);
            // u8g2->drawStr(0, 12 * 5, buf);
            u8g2->sendBuffer();
#endif
            delay(2000);
        }
        Serial.println();
    }
}

void led_Flash(uint16_t flashes, uint16_t delaymS)
{
    uint16_t index;
    for (index = 1; index <= flashes; index++) {
        digitalWrite(BOARD_LED, HIGH);
        delay(delaymS);
        digitalWrite(BOARD_LED, LOW);
        delay(delaymS);
    }
}
```

3.4.2 Ranging Slave

```
#include <SPI.h>
#include <SX128XLT.h>
#include "boards.h"
#define LORA_DEVICE DEVICE_SX1280                //we need to define the device we are using
//*******  Setup LoRa Parameters Here ! ***************
//LoRa Modem Parameters
const uint32_t Frequency = 2445000000;           //frequency of transmissions in hz
const int32_t Offset = 0;                        //offset frequency in hz for calibration purposes
const uint8_t Bandwidth = LORA_BW_0800;          //LoRa bandwidth
const uint8_t SpreadingFactor = LORA_SF8;        //LoRa spreading factor
const uint8_t CodeRate = LORA_CR_4_5;            //LoRa coding rate
const uint16_t Calibration = 11350;              //Manual Ranging calibration value

const int8_t TXpower = 10;                       //Transmit power used
const uint32_t RangingAddress = 16;              //must match address in master
const uint16_t  rangingRXTimeoutmS = 0xFFFF;     //ranging RX timeout in mS
SX128XLT LT;
uint32_t endwaitmS;
uint16_t IrqStatus;
uint32_t response_sent;

void led_Flash(unsigned int flashes, unsigned int delaymS);

void setup()
{
```

```
    pinMode(BOARD_LED, OUTPUT);
    led_Flash(2, 125);
    initBoard();
    Serial.begin(9600);            //setup Serial console ouput
    Serial.println("Ranging Slave Starting");
    delay(100);
    SPI.begin(RADIO_SCLK_PIN, RADIO_MISO_PIN, RADIO_MOSI_PIN);
    if (LT.begin(RADIO_CS_PIN, RADIO_RST_PIN, RADIO_BUSY_PIN, RADIO_DIO1_PIN, LORA_DEVICE)) {
        Serial.println(F("Device found"));
        led_Flash(2, 125);
        delay(1000);
    } else {
        Serial.println(F("No device responding"));
        while (1) {
            led_Flash(50, 50);                                  //long fast speed flash indicates
device error
        }
    }
    // The function call list below shows the complete setup for the LoRa
    // device for ranging using the information
    LT.setupRanging(Frequency, Offset, SpreadingFactor, Bandwidth, CodeRate, RangingAddress,
RANGING_SLAVE);
    LT.setRangingCalibration(11350);
     //override automatic lookup of calibration value from library table
    Serial.print(F("Calibration,"));
    Serial.println(LT.getSetCalibrationValue());   //reads the calibratuion value currently set
    delay(2000);
    u8g2->clearBuffer();
    u8g2->drawStr(0, 12, "Rang_Slave");
    u8g2->sendBuffer();
}

char buf[256];
void loop()
{
    LT.receiveRanging(RangingAddress, 0, TXpower, NO_WAIT);
    endwaitmS = millis() + rangingRXTimeoutmS;
    while (!digitalRead(RADIO_DIO1_PIN) && (millis() <= endwaitmS));
    //wait for Ranging valid or timeout
    if (millis() >= endwaitmS) {
        Serial.println("Error - Ranging Receive Timeout!!");
        led_Flash(2, 100);           //single flash to indicate timeout
    } else {
        IrqStatus = LT.readIrqStatus();
        digitalWrite(BOARD_LED, HIGH);
        if (IrqStatus & IRQ_RANGING_SLAVE_RESPONSE_DONE) {
            response_sent++;
            Serial.print(response_sent);
            Serial.print(" Response sent");
        } else {
            Serial.print("Slave error,");
            Serial.print(",Irq,");
            Serial.print(IrqStatus, HEX);
            LT.printIrqStatus();
        }
        digitalWrite(BOARD_LED, LOW);
        Serial.println();
    }
}

void led_Flash(unsigned int flashes, unsigned int delaymS)
{
    //flash LED to show board is alive
    unsigned int index;
    for (index = 1; index <= flashes; index++) {
        digitalWrite(BOARD_LED, HIGH);
        delay(delaymS);
        digitalWrite(BOARD_LED, LOW);
        delay(delaymS);
    }
}
```