University of the West of England

April 2018

Embedded IoT Development Kit

Alexander Collins

University of the West of England – Faculty of Environment and Technology – Department Of Computer Science and Creative Technology

COMPUTER SCIENCE (G400)

1 ABSTRACT

The Internet of Things (IoT) is still a young and evolving facet of technology. While certain areas of IoT have already become unanimous within our society (e.g. Smartphones), others (i.e. the more typical IoT solution) still have hurdles to overcome before they're fully integrated into our world.

This project aims to overcome some of those hurdles by providing an easy-to-use Embedded IoT Development Kit that utilizes Grove Modules to collect data and the Microchip RN2483 LoRa module to wirelessly transport this data.

Developed for this project are:

1. *A C software library for the RN2483 module, with specific platforms supported on individual branches of a git repository.*
2. *A C++ framework for using Grove Modules, with specific platforms supported on individual branches of a git repository.*
3. *A simple IoT solution using both the RN2483 library and Grove Module framework above.*

Hopefully this development kit will make it simple to implement a basic IoT solution that uses Grove and LoRa for anybody looking to use it or port it to their own platform.

2 CONTENTS

1 Abstract 1
2 Contents 2
3 Table of Figures 4
4 Table of Tables 5
5 Introduction 6
5.1 Deliverables 6
5.2 What is 'LoRa'/'LoRaWAN'? 6
5.3 What are 'Grove Modules'? 7
5.4 Related Works 7
6 Research 8
6.1 The Internet of Things 8
6.1.1 How will IoT integrate into society? 8
6.1.2 Case Study: Intel's 'Retail Sensor Platform' (RSP) 9
6.1.3 IoT's contribution to society 9
6.2 'Things' - IoT Hardware 11
6.2.1 Microcontrollers 11
6.2.2 LoRa Modules 14
6.2.3 Sensors 14
6.2.4 Software Development 16
6.3 'Internet' - IoT networks 19
6.3.1 LPWAN Technologies 20
6.3.2 LoRa/LoRaWAN 21
6.3.3 Microchip RN2483 LoRaWAN compliance 25
7 Methodology 27
8 Requirements 29
9 Design 35
9.1 LoRa RN2483 Library 35
9.1.1 Use Cases 35
9.1.2 Use Case – Send/Receive data over LoRa 38
9.2 Grove Drivers 39
9.3 Demo 40
10 Planning & Testing 41
10.1 Sprints 41
10.2 Testing 42
11 Implementation 46

11.1 Stage 1 – Prototyping..46
11.1.1 Grove Modules..46
11.1.2 RN2483..46
11.1.3 Demo..47
11.2 Stage 2 – Platform Setup..49
11.2.1 UWESense..49
11.2.2 BBC Micro:Bit..50
11.2.3 The Things Network (LoRaWAN network)..50
11.3 Stage 3 – Final Implementation..52
11.3.1 RN2483 Library..52
11.3.2 Grove Framework..56
11.3.3 Demo..57
12 Evaluation..59
12.1 Problems & Failures..59
12.2 Improvements..60
12.3 Achievements & Successes..61
12.4 Future Work..61
12.5 Closing Statement..62
13 Bibliography..63
14 Appendices..66
14.1 Defining 'Internet of Things'..66
14.2 Previous Grove Designs..68

3 TABLE OF FIGURES

FIGURE 1. FROM DATA COMES WISDOM (EVANS, APRIL 2011) 10
FIGURE 2. UWESENSE MICROCONTROLLER BOARD 12
FIGURE 3. EMBEDDED SYSTEM SOFTWARE LAYERS (BARR & MASSA, 2006) 13
FIGURE 4. RN2483 BLOCK DIAGRAM (MICROCHIP, 2016) 14
FIGURE 5. STANDARD GROVE PINOUT (SEEED STUDIO, N.D.) 15
FIGURE 6. LPWAN TRICHOTOMY 19
FIGURE 7. WAN TECHNOLOGY RANGE VS DATA RATE (GLUHAK, 2017) 20
FIGURE 8. LORA SPECTRUM (UPCHIRP) BASIC TRANSLATION 22
FIGURE 9. LORA SPECTRUM CAPTURE (BW = 8, SF = 8, CR = 4/6, DE = 1) (SIKKEN, 2018) 22
FIGURE 10. LORAWAN TOPOLOGY 24
FIGURE 11. LORAWAN CLASSES 24
FIGURE 12 LORAWAN CLASS A DEVICE 25
FIGURE 13. LORAWAN PACKET HEADER DECONSTRUCTED 25
FIGURE 14. RN2483 COMMAND INTERFACE (MICROCHIP, 2017) 26
FIGURE 15. SPRINT PROCESS 28
FIGURE 16. LORA RN2483 LIBRARY USE CASES 35
FIGURE 17. AUTOBAUD() STATE MACHINE 36
FIGURE 18. INITMAC() STATE MACHINE 36
FIGURE 19. RN2483 JOIN PROCEDURE 37
FIGURE 20. SEND/RECIEVE DATA OVER LORA 38
FIGURE 21. GROVE FRAMEWORK DESIGN (FINAL IMPLEMENTATION) 39
FIGURE 22. INITIAL DEMO DESIGN FIGURE 23. FINAL DEMO DESIGN FLOWCHART 40
FIGURE 24. BBC MICRO:BIT RN2483 SHIELD 50
FIGURE 25. BBC MICRO:BIT RN2483 SHIELD (CONNECTED) 50
FIGURE 26. #IFNDEF GUARDS 53
FIGURE 27. RN2483 MASTER BRANCH I/O 54
FIGURE 28. RN2483 PLATFORM/MBIT BRANCH I/O 54
FIGURE 29. DATA TO ITS HEX REPRESENTATION IN ASCII 54
FIGURE 30. HEX REPRESENTATION OF DATA (IN ASCII) TO A STRING 55
FIGURE 31. EXAMPLE OF HOW EASY IT IS TO USE THE GROVE FRAMEWORK ON THE BBC MICRO:BIT BRANCH (FIRST LINE IS SPECIFIC TO MICRO:BIT) 56
FIGURE 32. GROVE FRAMEWORK CLASS LAYOUT 56
FIGURE 33. DEMO CONSOLE LOG (BEFORE & AFTER SENDING DATA) 58
FIGURE 34. INITIAL DESIGN (BEFORE STAGE 1) 68
FIGURE 35. SECOND DESIGN (AFTER STAGE 1) 68

4 Table of Tables

Table 1. Standard Grove cable pinouts 16
Table 2. LoRa and LoRaWAN on the OSI model 21
Table 3. Project Requirements Table 34
Table 5. Sprint Schedule 42
Table 6. Test Results 45

5 INTRODUCTION

5.1 DELIVERABLES

This project aims to create an Embedded IoT Development Kit, accompanied with a demonstration of its usage. The target platform that will be used for demo solution will be the UWESense microcontroller (MCU) board. The development kit will include:

- A software library for the Microchip RN2483 LoRa module.
- Software drivers for a set of Grove Modules.
- A simple demo that illustrates this kit's usage.

The UWESense is a custom microcontroller board that uses an STM32F401 MCU, which is communicated to through the STM32 Hardware Abstraction Layer (HAL). It has a Microchip RN2483 module on-board and several Grove ports, making it ideal for this project.

The demo of this development kit will use a grove module to collect data and transport it using a Microchip RN2483 module. The data will be communicated to the UWESense via one of the developed Grove drivers. The data will then be sent to a local LoRa gateway via the RN2483 module, using the developed library. Finally, the data will be displayed on a client machine to assert the data has been successfully transported.

The goal of this project is to provide a development kit that is functional, easy to use, and provides full access to the functionality of the RN2483 and Grove Modules. The demo is intended to simply prove the LoRa library and Grove Modules work, but hopefully will show how to use them as well.

5.2 WHAT IS 'LORA'/'LORAWAN'?

LoRa is a wireless technology, considered to be an LPWAN (Low-Powered Wide-Area Network). LPWANs are a new subset of WAN technologies aimed at IoT solutions that the industry is currently in the process of standardising the usage of. Despite the number of competitors, LoRa appears to be one of the main LPWAN technologies with traction – this is in part due to the LoRaWAN protocol, an open-source specification that defines the usage of LoRa, maintained by the LoRa Alliance (a body companies interested in applying the technology).

Due to LoRa being a newer technology, there hasn't been much development for separate LoRa modules outside of popular/standardised platforms (e.g. Arduino). Another reason for this is the lack

standardisation in the embedded world[1], i.e. the module itself has a few libraries for platforms like the Arduino, but they're not portable to other platforms.

5.3 What are 'Grove Modules'?

Grove modules are a standardised brand of module for embedded microcontrollers, developed by seeed. These modules are specifically designed for IoT; the set of drivers will only cover a subset of Grove modules, as the total number of Grove modules is ridiculous. The popularity of these modules is born from their focus standardisation, ease-of-use and dedicated support to beginner platforms. seeed's standardisation of Grove Modules *does* have the downside that they require specific ports, which are designed so that they only fit Grove pinouts[2].

5.4 Related Works

There are several other works related to this project. None of them render the project useless though.

There have already been libraries developed for the RN2483 module, which are publicly available on GitHub[3]. Unfortunately, *all* these libraries are written for the Arduino platform, consequently these libraries are not portable to other target platforms (such as the UWESense).

Grove Modules have boilerplate code provided by the manufacturer (seeed) for *most* of their modules. This code is only exemplary though, their purpose is simply to display the logic required to communicate with the module – they do not provide users with a framework for using them.

[1] This is simply a side-effect of having so many options when picking a platform for an embedded solution.
[2] The physical dimensions of a Grove port don't fit most standard GPIO cables.
[3] E.g. https://github.com/jpmeijers/RN2483-Arduino-Library

6 RESEARCH

6.1 THE INTERNET OF THINGS

"There will be 25 billion devices connected to the Internet by 2015 and 50 billion by 2020." (Evans, April 2011)

If you've ever seen a talk related to the 'Internet of Things', you've likely heard the above quote (or similar) before. Unfortunately, the number of devices that gets quoted seems to vary anywhere between "*5.2 billion units in 2017*" (Meulen, 2017), to "*1 Trillion connected devices* [by 2015]" (Iwata, 2012); this confusion is a by-product of the term they relate to: 'Internet of Things' (IoT). IoT has "*no clear, single definition*" (Google, 2017); according to Uckleman, the term is "*not well defined and has been used and misused as a buzzword in scientific research as well as marketing and sales strategies*" (Dieter Uckelmann, 2011). "*definitions are crucial warrants*[4]" (Wayne C. Booth, 2008), ergo 'Internet of Things' requires a clear definition: "*A network of interconnected, uniquely addressed objects, that communicate and interact intelligently across the internet.*" (See **14.1**).

6.1.1 How will IoT integrate into society?

Even without a globally accepted and clear definition, IoT is going to become (and to some degree already is becoming) a large part of our society, which will undoubtedly "*play a leading role in the near future*" (Atzori, et al., 2010). This ubiquity is another by-product of how loosely defined the term 'Internet of Things' is[5] and also in-part due to the recent "*potential of a seemingly endless amount of distributed computing resources and storage*" (Miorandi, 2012) provided by the advent of 'Cloud Computing'.

Other factors, such as the push from large corporations in recent years has also heavily contributed. Consequently, "*The Internet-of-Things is emerging as one of the major trends shaping the development of technologies in the ICT sector at large*" (Jayavardhana Gubbi, 2013), this major trend is already coming to fruition: most large corporations have created business groups to introduce IoT solutions and technology on a larger scale.

As stated, there's been a recent push from corporations, a result of the IoT market being "*expected the grow 35% per year to $16 billion by 2020.*" (Verizon, 2017); how easily IoT integrates into other market technologies that are becoming increasingly important is also a large factor in this push. Technologies such as: Cloud Computing, which Amazon and Google are lead providers in; Artificial Intelligence (AI) – which has already been introduced to industries through companies like IBM (with

---

[4] "*A warrant is a statement that connects a reason to a claim.*" (Wayne C. Booth, 2008)

[5] The 'Things' part of 'Internet of Things' really does encompass a lot of different things. See https://gizmodo.com/15-idiotic-internet-of-things-devices-nobody-asked-for-1794330999

their Watson AI); and the growing importance of wireless networking technologies, like those facilitated by AT&T; etc. (Meola, 2016) all integrate into IoT.

6.1.2 Case Study: Intel's 'Retail Sensor Platform' (RSP)

As a case study of one of a large corporate IoT solution, and an example of what a real-world IoT solution looks like, we'll review Intel's 'Retail Sensor Platform' (RSP) product, which is *"an RFID-based system designed to make retail radio frequency identification deployments easier, as well as enable inventory tracking to be performed in real time."* (Swedberg, 2015).

This is arguably an exemplary IoT solution today - it demonstrates each step in today's model: the 'Thing'/'Device', 'Gateway', 'Cloud' and how it all ties together to *"solve today's environmental challenges and benefit customers and society at large"* (CERP-IoT - Sundmaeker, et al., 2010). Intel's RSP provides this benefit by giving retail stores knowledge of *"what inventory they have in-store"*, *"alerts when inventories are low"* and where *"misplaced items"* (Intel Corporation, 2015). Improvements like these can provide massive boosts to efficiency, value and provide stores with more data for improving customer experience. The data provided by solutions like this are often processed by *"enterprise-level Big Data analytics and visualization"* (Intel Corporation, 2015), which in the long-run will be an even more crucial factor in the technology's contribution to society.

One important thing to note about this solution, is that its RSP sensors require a cable link to the gateways. This is a serious limit to the scalability of the solution and an example of a situation where an embedded wireless technology would be an appropriate addition.

6.1.3 IoT's contribution to society

Unfortunately, *"Most enterprise IoT projects are in the proof of concept or pilot phase, not in production"* (Verizon, 2017), so the earlier stated 'benefit to society' that IoT has the potential of is still untapped. The potential is sourced from how *"IoT dramatically increases the amount of data for us to process"* (Evans, April 2011).

This contribution is a result of how our society uses data to progress. **Figure 1** demonstrates the *"direct correlation between the input (data) and output (wisdom)."* (Evans, April 2011). When considered alongside the scale data that will be provided by IoT, you get a clear understanding of why IoT is considered to be the *"third and potentially most "disruptive"6 phase of the internet revolution"* (CERP-IoT - Sundmaeker, et al., 2010).

As the provider of 'Big Data', IoT will be one of the cornerstone technologies at the forefront of this revolution.

---

[6] *"The term "disruptive technology" was coined by Clayton M. Christensen and introduced in his 1995 article Disruptive Technologies: Catching the Wave, which he co-wrote with Joseph L. Bower. Ref. Harvard Business Review, January-February 1995."* (CERP-IoT - Sundmaeker, et al., 2010)

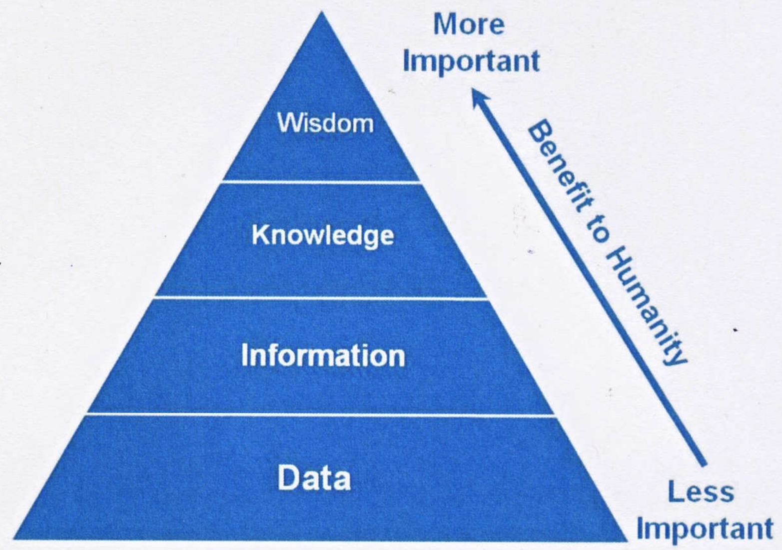


Figure 1. From Data comes Wisdom (Evans, April 2011)

6.2 'Things' - IoT Hardware

6.2.1 Microcontrollers

6.2.1.1 How do Microcontrollers fit into IoT?

Microcontrollers (MCUs) are one of the key technologies used in IoT solution. They are what make the 'Things' smart. You'll likely have noticed how often the word 'smart' appears in marketing for IoT.

Officially, a 'smart object' is *"defined by the IPSO Alliance as being small computers with a sensor or actuator and communication device, embedded in objects"* (CERP-IoT - Sundmaeker, et al., 2010). *"A microcontroller is a processor with memory and a whole lot of other components integrated on one chip."* (Gridling & Weiss, 2007). Essentially: a very minimal and stripped-down computer system designed to run a minimal number of simple programs. Microcontrollers offer IoT developers: *"Small size, low-power consumption, and flexibility"* (Sign, 2008), helping meet a lot of Atzori's design objectives for IoT solutions[7].

6.2.1.2 MCU Design

The flexibility of microcontrollers is one of the major reasons they're popular in IoT; this flexibility comes from the wide variation in: memory type, speed/clock frequency, GPIO pin-count, peripherals, connectivity, power consumption, size, cost, etc. It allows for a developer to design a microcontroller to perfectly fit their solution. Unfortunately, designing a custom microcontroller has the downside that *"decisions require intricate knowledge of the system* [software], *which a hardware designer usually does not have"* (Ernst, et al., 1993). Not only this, but when experts develop a custom microcontroller to perfectly fit their use case, it often means that the *"hardware-software codesign potentially has a much higher impact"* (Ernst, et al., 1993).

The potential good is that this can result in a much better performance; the potential bad being that it requires the person(s) working on the hardware and software to be *very* in-sync, and knowledgeable of each other's domain. For a short-term IoT project, with a small technical team, this isn't an issue. Where a project is larger (such as the earlier case study) and/or going to require handing-over in the future[8], it becomes an unsustainable problem.

The solution to this is pre-designed MCUs and MCU boards. More accessible microcontrollers are often more desirable; various companies design and sell MCUs/MCU boards with varying philosophies. For example, Arduino make their MCU boards as user-friendly as possible with a focus on education. They are consequently very good as prototyping and learning platforms; other companies, such as ST, design MCUs with a much more expert approach and allow for lower-level access, erring towards the

[7] *"energy efficiency"*, *"scalability"*, *"reliability"*, and *"robustness"* (Atzori, et al., 2010)

[8] The average supported lifetime of an IoT solution is 7 years.

approach discussed by Ernst, et al., 1992. These companies attempt to alleviate the pains of hardware-software co-design by providing drivers and Hardware Abstraction Layers (HAL)'s. These at least provide some kind of shared knowledge pool for developers to share.

6.2.1.3 UWESense MCU Board

This project focuses on the UWESense microcontroller board. It demonstrates an interesting middle-ground between custom-designed microcontroller boards and those pre-designed by manufacturers; the actual board has a custom design, but uses pre-designed components. Ergo, it still uses the same generic Drivers and interfaces that other developers might've interacted and have familiarity with.

The board design itself has a lot of the standard trappings of an IoT microcontroller board – namely: a lot of peripheral ports and a low-power MCU. The 'Grove pinouts' on the board are specifically designed to fit seeed's Grove System cable pinout (**6.2.3**). There are also two JTAG programmer interfaces that are used to program and debug their respective components (the Bluetooth and MCU chips); the board also has wireless capabilities – Bluetooth and LoRa, which is currently less standard[9].

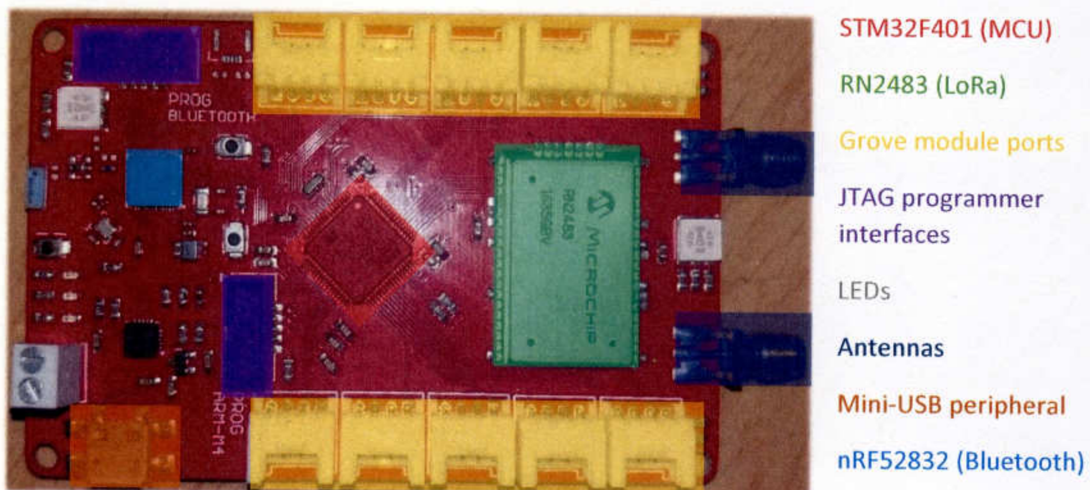


Figure 2. UWESense Microcontroller Board

[9] Wireless capabilities on MCU boards will likely be much more common in the future, however as of today MCU wireless technologies are still competing to become the standard for their respective use cases.

6.2.1.4 STM32F401 MCU

In this case, the MCU on the board is an STM32F401. It's part of the *"STM32 family of 32-bit Flash microcontrollers based on the ARM® Cortex®-M processor"* (ST, n.d.). This MCU is a perfect example of the kind of size and power that's typical in IoT; the chip itself is only *"3x3mm"* (ST, n.d.) and provides *"the performance of Cortex®-M4*[10] *core with floating point unit, running at 84 MHz"* (ST, n.d.). Of course, an MCU can't just be small and provide power – it needs to be integrated onto boards (such as the one above) to be of any use. To do this it has:

- *3x USARTs running at up to 10.5 Mbit/s*
- *4x SPI running at up to 42 Mbit/s*
- *3x I²C*
- *1x SDIO*
- *1x USB 2.0 OTG full speed*
- *2x full duplex I²S up to up to 32-bit/192KHz*
- *12-bit ADC reaching 2.4 MSPS*
- *10 timers, 16- and 32-bit, running at up to 84 MHz*

(ST, n.d.).

For a developer to interact with the MCU, they will (in most cases) use generic drivers and read/write to the chip via the JTAG interfaces. In the case of the STM32F401, a HAL[11] and set of Low Layer (LL) drivers has been developed by the manufacturer (ST) for interacting with the component: *"It is directly built around a generic architecture and allows the built-upon layers, such as the middleware layer, to implement their functions without knowing in-depth how to use the MCU."* (ST, 2017). This is a typical example of the kind of abstraction to avoid the problem of hardware-software co-design.

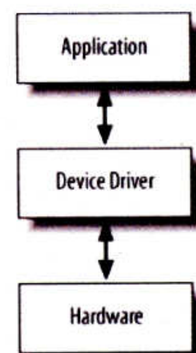


Figure 3. Embedded system software layers (Barr & Massa, 2006)

[10] The ARM® Cortex®-M series of processors are one of the most popular in embedded.

[11] A Hardware Abstraction Layer is a software layer (usually composed of a set of drivers) designed to *"hide the hardware completely"* (Barr & Massa, 2006)

6.2.2 LoRa Modules

6.2.2.1 RN2483

The RN2483 chip enables one of the two wireless capabilities of the UWESense – LoRa (the 2nd being Bluetooth via the nRF52832). Unlike the nRF52832, the RN2483 chip isn't directly accessed via a JTAG interface but is integrated using the STM32F401's UART I/O capabilities. The RN2483 has a *"dedicated UART interface to communicate with a host controller"* (Microchip, 2016). The device itself is the very specifically designed type of Microcontroller discussed earlier in **6.2.1.1**, running its own software to allow external hosts to interact with it; the chip is designed to be integrated with more powerful host MCUs, but still contains all of the components any usual MCU would (see **Figure 4**). Interaction is done with the RN2483 via its *"ASCII Command Interface"*, which is a set of commands the module listens for and responds to accordingly. The LoRa capabilities of this module will be discussed further in **6.3.3**.

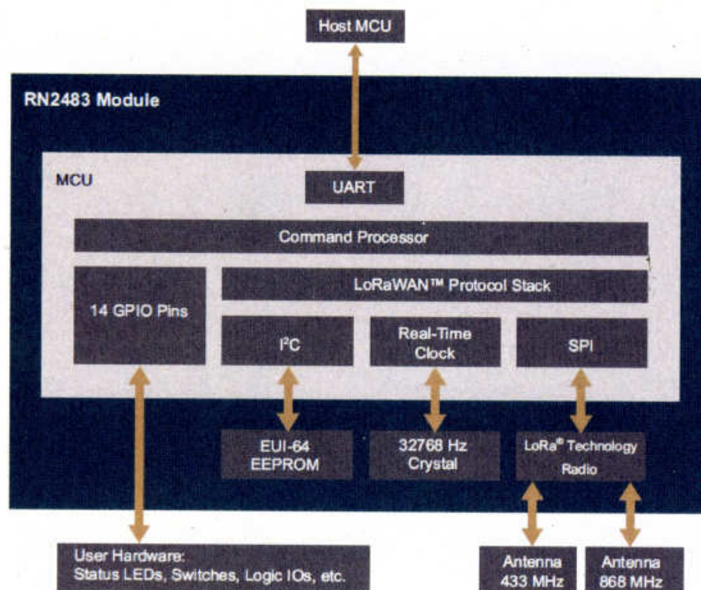


Figure 4. RN2483 Block Diagram (Microchip, 2016)

6.2.3 Sensors

Linked closely to Microcontrollers (in IoT), are sensors - sensors are how the solutions collect information about the objects and/or environments being monitored[12]. In its current form sensors are one of the core pillars of IoT, they provide raw data that (noted in **Figure 1**) is processed into wisdom that will ultimately contribute to society.

Sensors convert readings (such as temperature, light, etc.) into signals that can be processed by a computer; most sensors will send their readings to whatever their output pin is plugged into as long as they're powered. This simplicity results in low power requirements and consequently allows them to be powered by a Microcontroller without draining too much of its own battery. A standard (simple)

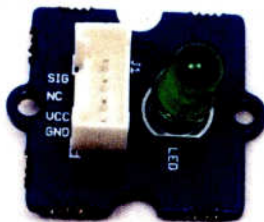
[12] Even the earlier Case Study (**6.1.1**) used an RFID sensor.

sensor would only require a maximum of 3.3v - 5v[13]. This allows for one MCU to provide power to multiple sensors and still last for years on battery power - you might have noticed the UWESense MCU board has enough ports for 10 Grove modules.

6.2.3.1 Grove System

There are several peripheral ports on the UWESense board are specifically designed to fit seeed's Grove System cable pinout (See **Table 1**). The Grove System is designed to be a simple, "*modular, standardized connector prototyping system*" (seeed studio, n.d.), these attributes are attained through:

1. A standardized cable pinout (**Table 1**)
2. Standardized module sizes
3. Extensive user-friendly wiki documentation of each individual module
4. Manufacturing of Grove shields[14] that can be easily attached to supported MCU boards and provide more I/O, as well as port for the standard cables in 1
5. Extensive support for the beginner IoT boards (such as Arduino or Raspberry Pi)
6. There seems to be a Grove module for every sensor/actuator/display type imaginable



The standardized Grove cable pinout is likely the biggest ease-of-life factor that contributes to their population. It stops users mixing up pins, like VCC and GND (which can result in killing a module), as well as making the pins clearer (which removes a large pitfall that can cause users to spend hours debugging before they realise they simply mixed up cables).

Figure 5. Standard Grove pinout (seeed studio, n.d.)

[13] This is why most MCU boards have 3.3v and 5v pinouts.
[14] Add-on modules for MCUs.

Table 1. Standard Grove cable pinouts

Pin	Colour	Connector Type	Function	Note
1	Yellow	Digital	Dn	Primary Digital Input/output
2	White	Digital	Dn+1	Secondary Digital Input/Output
1	Yellow	Analogue	An	Primary Analogue Input
2	White	Analogue	An+1	Secondary Analogue Input
1	Yellow	UART	RX	Serial Receive
2	White	UART	TX	Serial Transmit
1	Yellow	I2C	SCL	I2C Clock
2	White	I2C	SDA	I2C Data
3	Red	ALL	VCC	Power for Grove Module, 5V/3.3V
4	Black	ALL	GND	Ground

6.2.4 Software Development

Grove modules are made easy to use through their standardisation; however, they still require software to read from them. While the supported starter platform users (such as Arduino) are even provided with example code to read/write to/from the modules, other more non-standard boards (such as the UWESense) interface to the modules through their own methods.

As previously stated, microcontrollers often come with generic drivers that helps provide developers with this hardware interface. Unfortunately, these drivers don't make development on the platform as high-level as something like an operating system[15] and *"due to the pitfalls and peculiarities of direct hardware access, embedded software developers actually face quite different challenges than those working in higher level environments"* (Gridling & Weiss, 2007). Therefore, said developers still require (at least a general) knowledge of the hardware and expertise in the relevant language.

6.2.4.1 Assembly vs C

"C is the closest thing there is to a standard in the embedded world" (Barr & Massa, 2006)

C is the most popular language for embedded software development, it *"can satisfy the requirements of both high memory efficiency and high performance owing to good compilers"* (Inoue, et al., 1998); 10-20 years ago, Assembly was the most popular – this is back when direct hardware access was much more standard (even in OS') and before compilers were able to output binary code that could match

[15] Albeit, some platforms (again Arduino) do make it incredibly easy by comparison.

the optimisation of an Assembly programmer[16]. Instead, Assembly is used *"primarily as an adjunct to the high-level language, usually only for those small pieces of code that must be extremely efficient or ultra-compact, or cannot be written in any other way."* (Barr & Massa, 2006).

While C is the most popular, many cite problems with it – one of the most common being the ability for a developer to shoot themselves in the foot with the ability to directly access memory through pointers and memory handling functions, however for an embedded developer, where memory is limited, this should be considered a good thing (it just requires they be more careful).

Another, and more interesting problem is the *"limited word-length support. Only a few data sizes, which are 8 bits, 16 bits, and 32 bits[17], are supported in C. It is insufficient for many applications such as audio processing in which 24-bit data is typical"* (Inoue, et al., 1998). While *"32-bit data types may be used instead"*, this costs memory, which is precious in an embedded environment. Workarounds this problem have been developed since, however they're often hacky and compiler-specific.

So why has C become more popular than Assembly in embedded? One of the main reasons is that C's higher-level. Where developers are required to learn a new set of instructions for different architectures in Assembly, in C this intricacy is left for the compiler to deal with, which means that a C developer can move between environments with relative ease. This isn't to say that C has made Assembly redundant – however where Assembly is required, it's often written in-line within C code.

6.2.4.2 C++ in Embedded

Becoming more common in embedded is C++. In years prior, C++ was generally considered too expensive and slow for embedded, to the point that *"In 1996, a group of Japanese processor vendors joined together to define a subset of the C++ language and libraries that is better suited for embedded software development."* (Barr & Massa, 2006). However, this subset (known as 'Embedded C++'), doesn't appear to have stuck around and was even looked upon negatively by the creator of the C++ language: *"To the best of my knowledge EC++ is dead (2004), and if it isn't it ought to be."* (Stroustrup, 2017).

The fact that *"the syntactical differences between the languages* [C and C++] *have little or no runtime cost associated with them."* (Barr & Massa, 2006) should indicate that C++ can be as suitable as C is for embedded, however many state that *"It is generally agreed that C++ programs produce larger executables that run more slowly than programs written entirely in C"* (Barr & Massa, 2006).

Barr & Massa wrote in 2006 that features such as *"templates, virtuals, exceptions and runtime type identification"* (Barr & Massa, 2006) were too expensive – but an ISO/IEC technical report on the

[16] Even today a good Assembly programmer could beat a compiler (although the time it takes might not be worth it)

[17] Support for native, portable 64-bit variables has been added since this paper was published.

performance of C++ from the same year noted that *"there is no additional real-time overhead for calling function templates or member functions of class templates"* (ISO/IEC, 2006). However, regarding Inheritance noted:

- (Single Inheritance) *"Converting a pointer to a derived class to a pointer to a base class will not introduce any run-time overhead in most implementations"* (ISO/IEC, 2006)
- (Multiple Inheritance) *"Converting a pointer to a derived class to a pointer to a base class might introduce runtime overhead"* (ISO/IEC, 2006)
- (Virtual Inheritance) *"Converting a pointer to a derived class to a pointer to a virtual base will introduce run-time overhead"* (ISO/IEC, 2006)

And that features such as *"Dynamic Casts"*, *"Dynamic Memory Allocation"* and *"Exceptions"* *"are more likely"* or *"will introduce overhead"* (depending upon their implementation) (ISO/IEC, 2006).

Consequently, the best conclusion to draw from this is that while it's perfectly possible for C++ to be used for embedded software, it requires well-written C++ for it to be implemented without issues. An assumption to the reasons why C is still the more popular in embedded software than C++ would likely come down to C being a relatively small language (making porting a compiler much easier) and more fail-safe than C++ because of the potential overhead (depending upon implementation).

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off" (Stroustrup, 2017)

6.3 'INTERNET' - IOT NETWORKS

As established, IoT is going to be responsible for *a lot* of devices. The *"Internet of Things, calls for open, scalable, secure and standardised infrastructures which do not fully exist today"* (Dieter Uckelmann, 2011), one of the main challenges in creating this infrastructure, is IoT's broad scope. Luckily this variation in technology has already started to sprout into several standards developed by organisations like the IEEE, IEFT, ITU, etc.

A study on the current 'ecosystem' of IoT protocols and standards found 13+ Data Link protocols, stating that *"their main differences and usefulness is in IoT medium access"* (Tara Salman, 2017). Each of these protocols have different advantages and disadvantages, allowing for many of them to co-exist, as the use cases they apply to differ widely (where different amounts of data need to be sent at different speed, across different distances, etc.).

Each use case has a preferred technology though. As of 2018, some of the most popular co-existing standards are: Bluetooth, Cellular, WiFi, Zigbee, etc. There is a use case that has, for several years, been left unfilled by these technologies though. This use case is currently one of the newest trends IoT and will soon become a standard in the IoT model; that is, a low-power, long-distance wireless technology.

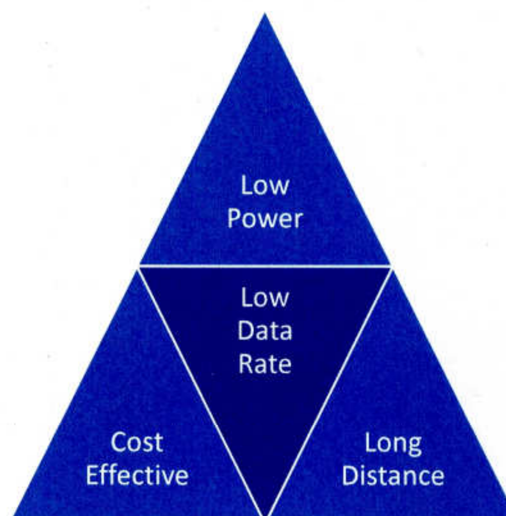


Figure 6. LPWAN Trichotomy

6.3.1 LPWAN Technologies

"The first thing to understand is that LPWAN (Low Powered Wide Area Network) is NOT a standard"

(Leverege, 2016)

LPWAN is a subset of WAN technology that captures a set of characteristics which make it ideal for a large group of IoT applications that currently piggyback on other networks (such as cellular), which over-provide what these use cases require.

There are more reasons that LPWANs are required for these applications beyond efficiency though – for example, cellular *"costs and power demands don't make them acceptable ... where battery powered operation is essential and equipment and connectivity costs must be low"* (Gluhak, 2017); Mesh Networks (such as Zigbee), which are *"only useful at medium distances and do not have the long-range capabilities ... More importantly, mesh networks are not battery efficient."* (Leverege, 2016).

Figure 6 shows the three characteristics that define LPWAN technology and fill the gaps of existing IoT wireless technologies. Unfortunately, these characteristics also inherit downsides: *"A physical limitation to achieve low power and wide range is a small data size."* (Leverege, 2016), the low data rate is illustrated in **Figure 7**. This makes the most ideal use case for LPWAN technologies *"situations where devices need to send small data over a wide area while maintaining battery life over many years"* (Leverege, 2016).

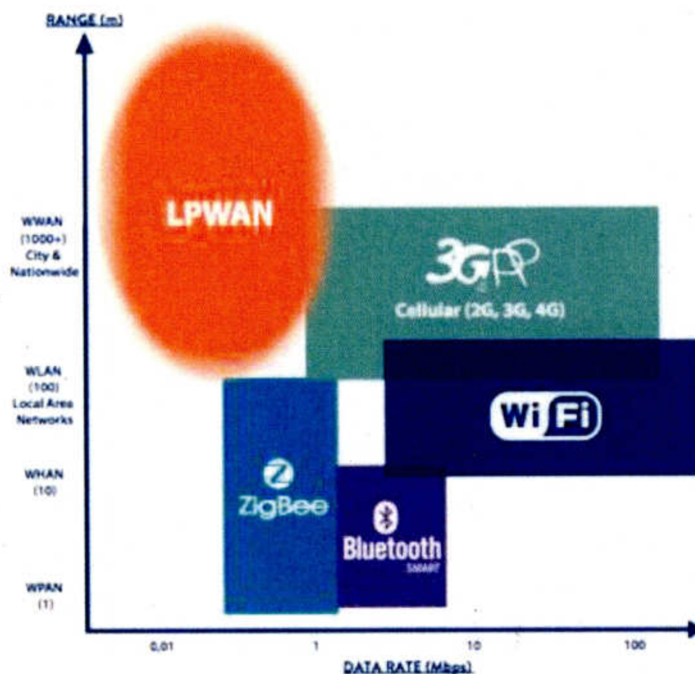


Figure 7. WAN Technology Range vs Data Rate (Gluhak, 2017)

6.3.1.1 *Unlicensed ISM Band*

The current LPWAN market is in its infancy, having only started to trend in the last few years. There are, however, still a lot of competitors despite this: SigFox, LoRa/LoRaWAN, Weightless, 6LoWPAN, etc. Each of these are unique and have different pros and cons; one thing they all have in common is that *"Major LPWAN technologies currently use a sub-GHz unlicensed ISM band"* (Leverege, 2016). The use of the unlicensed ISM band frequencies is arguably what provides LPWANs with the characteristics that define them.

One of the main appeals of the sub-GHz band is that there are *"no licensing is required for public access"* (Dongare, et al., 2017). Consequently, it's free to use, which maintains the *Low Cost* of LPWANs. As an example of the antithesis, operating on licenced frequencies (e.g. cellular) costs exorbitant fees[18], which are designed to avoid the heavy interference found on the unlicensed frequencies. Interference on unlicensed sub-GHz frequencies is another inherent problem of using this band that LPWANs are required to overcome. Many technologies use *"frequency hopping to minimize the interference"* (Tara Salman, 2017).

Another reason sub-GHz frequencies are used comes down to its ability to *"penetrate more deeply into structures"* and *"transmit over distances as long as 10 km with the same power consumption (or less)"* (Dongare, et al., 2017), this provides LPWANs with their *Long Range* characteristic.

6.3.2 LoRa/LoRaWAN

As mentioned, there are a few key players emerging in the LPWAN market, such as SigFox, NWave, Weightless, etc. LoRa/LoRaWAN is one which *"has received a lot of attention in recent months from network operators and solution providers"* (Ferran Adelantado, 2017). The technology aims to suit use cases with *"power saving, low cost, mobility, security, and bidirectional communication requirements"* (Tara Salman, 2017). However, it's important to note that LoRa and LoRaWAN are two separate entities that only encapsulate an LPWAN stack when combined.

Table 2. LoRa and LoRaWAN on the OSI model

4. Transport	Reliable transmission of data segments between points on a network.	LoRaWAN
3. Network	Structuring and managing a multi-node network, including addressing, routing and traffic control	LoRaWAN
2. Data link	Reliable transmission of data frames between two nodes connected by layer 1	LoRa / LoRaWAN
1. Physical	Transmission and reception of raw bit streams over radio frequencies	LoRa

[18] In an auction, the FCC was selling frequency space in the US for up to $900,000,00. (Knight, 2016)

6.3.2.1 *LoRa*

"In an analogy to the OSI communication stack, LoRa radios define the first and second layers (Physical and Data Link) of LPWAN" (Dongare, et al., 2017). LoRa radios are provided by Semtech, who patented the *"Chirp Spread Spectrum"* (Hornbuckle, 2008) – the scientific technology behind how LoRa radios physically communicate over radio waves, with a focus on distance and reliability. Being patented of course means that 'LoRa' is a closed-source technology, ergo LoRa radios can only manufactured by Semtech. Despite this, there has been effort to reverse-engineer the LoRa technology[19] out of concern for PHY layer security. **Figure 9** and **Figure 8** demonstrate a *very* basic understanding of how LoRa communicates bits over radio waves in an uplink; each upchirp/downchirp 'wave' is of different lengths and combine to form a bitstreams.

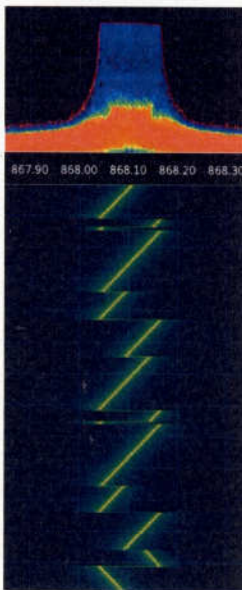


Figure 9. LoRa Spectrum capture (BW = 8, SF = 8, CR = 4/6, DE = 1) (Sikken, 2018)

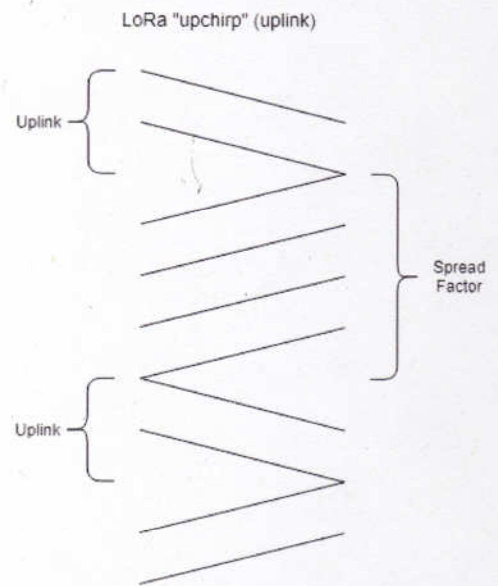


Figure 8. LoRa Spectrum (upchirp) basic translation

The physical LoRa chips that Semtech develop come in two forms (as of 2018): the SX12 series and the SX13 series. The difference between the two that one is designed to be used for end-nodes and the latter for gateways; the SX12 series are single-channel and much cheaper; the SX13 series are used in gateways and cost much more, however being multi-channel allows them to receive multiple LoRa packets simultaneously.

---

[19] See (Knight, 2016)

6.3.2.2 LoRaWAN

"Layers three and four (Network and Transport) are analogous to the LoRa Wide-Area Network (LoRaWAN) protocol." (Dongare, et al., 2017). LoRaWAN is the protocol that's used to regulate the usage of LoRa between networks and arguably one of that main reasons the technology has gained traction. *"The protocol and network architecture have the most influence in determining the battery lifetime of a node, the network capacity, the quality of service, the security"* (LoRa Alliance, 2015).

As previously noted, networks with a mesh topology are not battery efficient since each node needs to send data to each connected node/gateway; a lot of LPWAN protocols use a star topology to counter this, however this also means that the reliability of packets is much lower. LoRaWAN counters this by using a star-on-star topology (see **Figure 10**):

1. Each node sends its packets out, with no specific target.
2. Gateways in range pick up any LoRa packet received and verify it's a validity by checking the header.
3. If the packet is valid, it's forwarded via TCP/IP to the Network Server, which handles mitigation and verification of timestamps, etc.
4. After sending it to the application, the network server will the send any downlink response through the gateway considered "best" for replying to that node.

"the topology suggests they were really thinking about security when they designed it."

(Knight, 2016)

We can also discern from the topology that it was designed with the idea of using LoRaWAN as a public network. Whilst the protocol can very easily be used in a private network, a lot of the features that make it so scalable seem like overkill for anything private outside of enterprise solutions[20].

An example of LoRa working incredibly well in a public network is The Things Network, a public LoRaWAN network that's open-source and has coverage provided by its user's own gateways. This has proven to be a very successful enterprise and has also helped spread the availability of LoRaWAN to places which didn't have it before. A good example of this is in the UK where it's currently the only public network available.

Another feature of LoRaWAN that helps it scale so well are the two "join modes" that end-nodes can use to join a network: Activation-By-Personalisation (ABP) and Over-The-Air-Activation (OTAA). ABP is a more standard network join mode, where both the server and the end-node have matching keys.

[20] Which might be another reason LoRa/LoRaWAN is gaining so much traction; as established in **6.1**, enterprise solutions are a next step in the IoT market.

OTAA is a more interesting option that improves the scalability of network; where valid keys are exchanged during a join procedure.

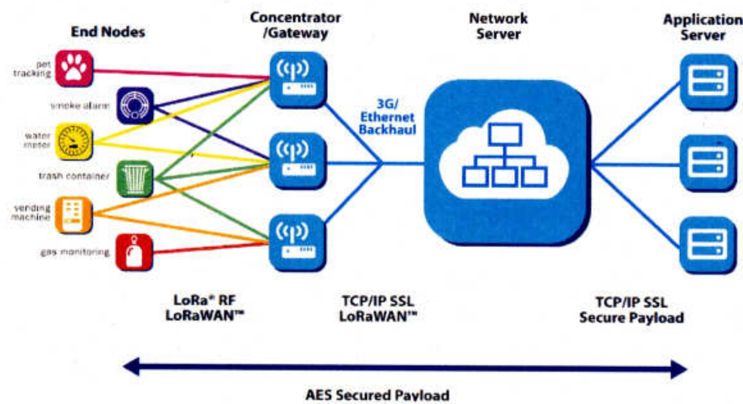


Figure 10. LoRaWAN topology

As well as having this topology, the LoRaWAN defines 3 classes of end-node devices: Class A, Class B and Class C. All classes provide bi-directional communication, however differentiate themselves in the amounts and frequentness they're able to receive data (which also has a big effect on their battery life).

- **Class A** – *"simple low-power sensor nodes that follow the so-called Aloha[21] protocol"* (Battle & Gaster, 2017).
    - 1 Uplink, followed by 2 short Downlinks[22].
- **Class B** – *"regularly poll for incoming messages (every 128 seconds)"* (Battle & Gaster, 2017).
    - Class A functionality + a scheduled polling window for Downklinks.
- **Class C** – *"listen continuously for incoming messages"* (Battle & Gaster, 2017).
    - Only close their Downlink window when an Uplink is in process[23].

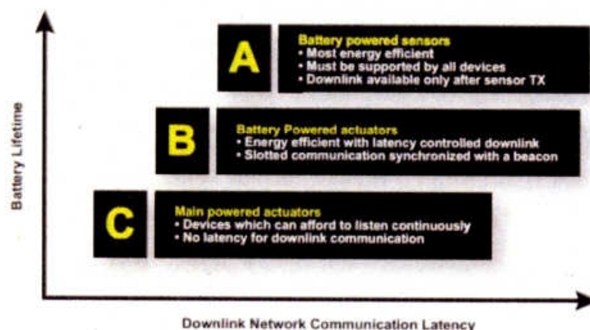


Figure 11. LoRaWAN Classes

[21] *"'Aloha' is not an acronym, but simply a way for asynchronous devices to announce themselves by sending a 'hello' message (the data packet)."* (Battle & Gaster, 2017)

[22] Uplink is the term used in the LoRaWAN protocol for data transmission, Downlink is used for receiving data.

[23] A main difference between Node radios and Gateway radios is that Node radios are cheaper and single-channel. LoRa Gateways argued as multi-channel Class C devices.

Currently, Class A devices are the most popular and widely used. This project will be using a Class A device simply because the LoRa Radio being used has a LoRaWAN layer of functionality that works using the Class A specification (see **Figure 12**).

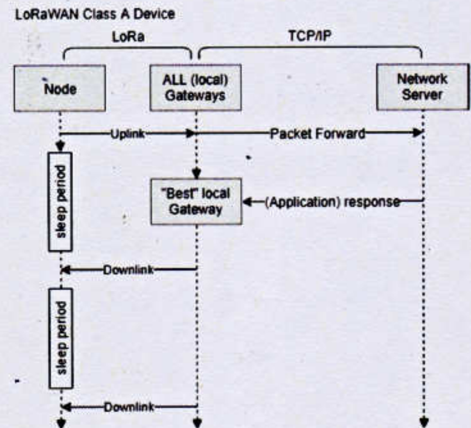


Figure 12 LoRaWAN Class A Device

6.3.3 Microchip RN2483 LoRaWAN compliance

As previously discussed, the Microchip RN2483 is the LoRa radio being used for this project. Except for the radios that allows the user directly access to device registers (e.g. the RFM95), LoRa radios will have a software interface that sets a lot of the header settings (within the PHYPayload) for the user. This is so that the radios can send a LoRaWAN compliant packet header without the user being required to know it inside out; the Microchip RN2483 specifically allows the user to set whether a tx packet is to be confirmed, it's payload and port.

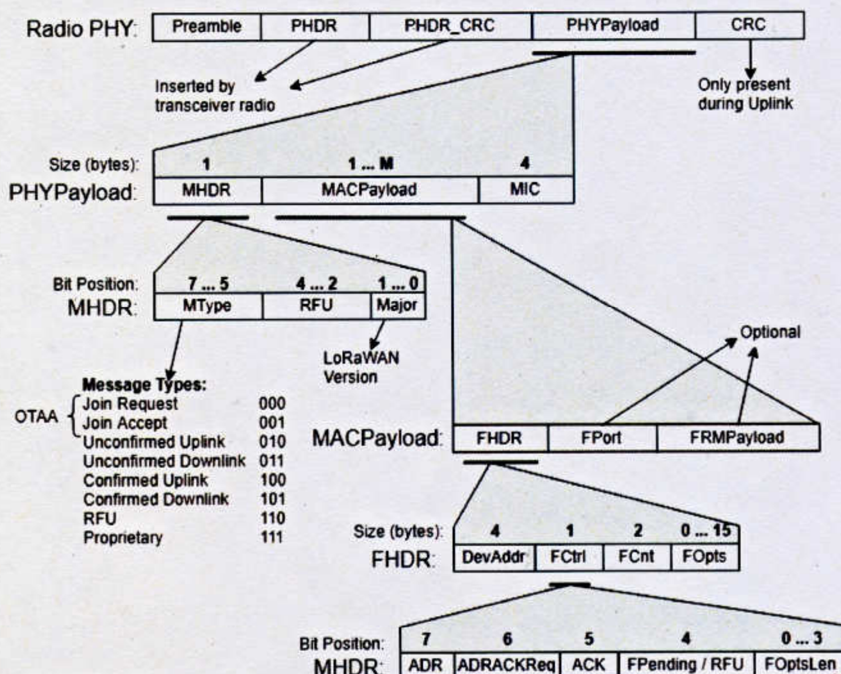


Figure 13. LoRaWAN Packet Header deconstructed

One interesting note about the Microchip RN2483 is that it allows users to break outside of the LoRaWAN compliance with *"radio"* and *"sys"* commands (see **Figure 14**). This project won't require this functionality, since it aims to create a LoRaWAN compliant library, however it's good to note that LoRaWAN isn't *required* to use LoRa radios and is implemented through the PHY payload.

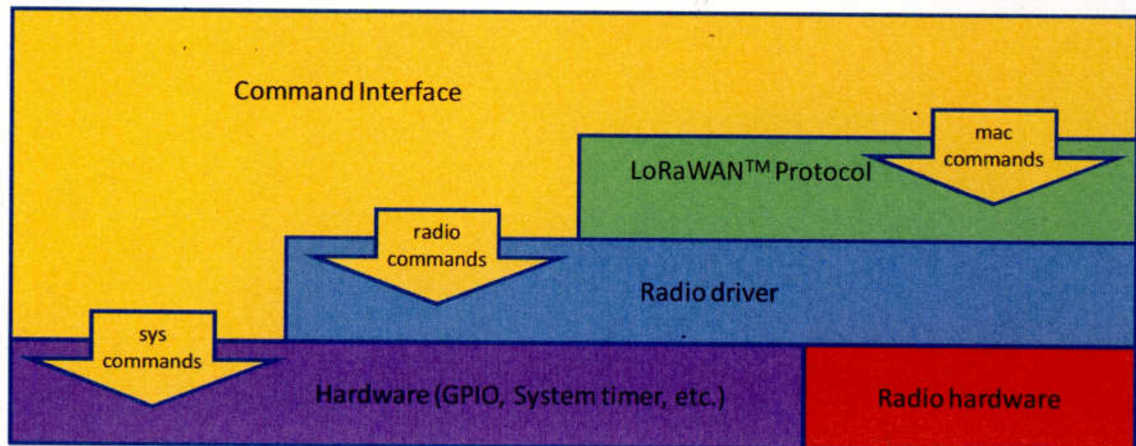


Figure 14. RN2483 Command Interface (Microchip, 2017)

7 METHODOLOGY

For this project, the AGILE methodology Test-Drive Development will be used. After review, TDD is a natural fit for embedded development:

- *Reduce the number of long target compile, link, and upload cycles that are executed by removing bugs on the development system.*
- *Reduce debug time on the target hardware where problems are more difficult to find and fix.*
- *Isolate hardware/software interaction issues by modelling hardware interactions in the tests.*
- *Improve software design through the decoupling of modules from each other and the hardware. Testable code is, by necessity, modular.*

(Grenning, 2011).

Specifically, when working with hardware in embedded software development, it's common to run into problems caused by a hardware specific issue that aren't immediately obvious and consequently act as a road-block until the developer is able to solve the issue. Hopefully, TDD should help mitigate this problem by solving and verifying those issues as they're occur (instead of discovering them later down the road where they become a larger issue). It'll also mean that any software successfully written can be considered "finished". As an example of why this will be being important for this project in particular: basic tx/rx communication with the relevant modules is going to be a requirement before any other code is written for them, therefore the tx/rx communication must be written, tested and validated so that further development and testing can continue. With the tests and validation, it should also mean that the established tx/rx communication is as reliable as the project requires and won't create any issues further down the road.

Specifically, TDD will be used in combination with a few sprints that should allow for achieving milestones which will complete the project as required[24]. Each sprint will be a week long and have a defined goal. The week following the sprint will be used to test the code written during the sprint, these tests will be prioritised on how important is to pass them (*must*, *should*, *could*).

The tests prioritised as *must* are required to pass successfully for the sprint to be considered successful; any *should* or *could* tests will be considered bonus and extra validation of the unit, although *should* tests will be added to a backlog and added back into future related sprints. The testing

[24] Even if it does not allow for completion of the project, it will provide a clear marker on progress

shouldn't take longer than a week at most, once it has been complete, the review process will begin. In the review process, any failed *must* test cases will be reviewed and resolved.

As Bob Martin states in his first of the three 'Laws of TDD': "*Do not write production code unless it is to make a failing unit test pass.*" (martin, n.d.); The review stage will only be moved on from once all the *must* tests have been passed, regardless of whether another sprint is scheduled. If all *must* test cases are passed or if there is leftover time during the review process, the code will be reviewed for improvement. If the next sprint is scheduled while there are unresolved *must* test cases, then other solutions will be considered.

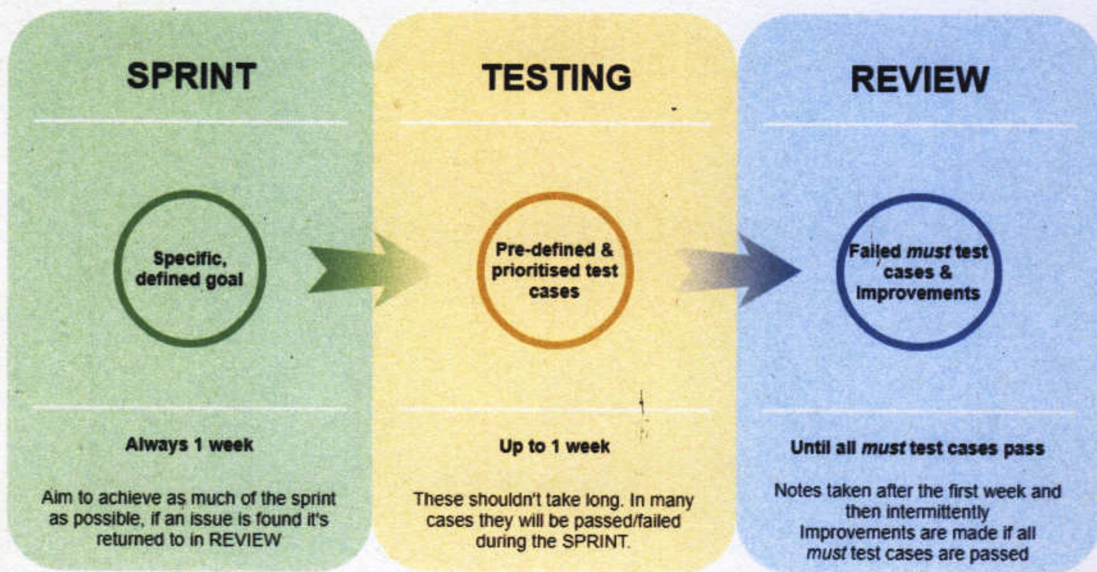


Figure 15. Sprint Process

8 Requirements

ID #	Requirement Description	E.G. use case / Justification	Functional / Non-Functional	User / System	Priority
0	**Develop a software library for the RN2483 LoRa module** This is one of the main goals of the project. The library will provide full functionality of the RN2483 LoRa module.	Software running on a microcontroller, using this library to talk to a connected RN2483 module to send data over LoRa to a gateway.	Functional	User	Must Have
1	**Develop a set of Grove Module drivers** This is one of the main goals of the project. The drivers provided will allow for full functionality of a set of Grove Modules	Software running on a microcontroller, using one of these drivers to receive data from the corresponding connected module.	Functional	User	Must Have
2	**Develop a demo to provide example usage of #0 and #1** This is to prove **#0** and **#1**, as well as provide an example use case. It will demonstrate basic functionality of the LoRa library and reading from a sensor with the Grove Sensor drivers.	Any developers figuring out how to use **#0** or **#1** might use this as demo to see how to properly interact with them.	Functional	User	Must Have
3	**Develop a demo to fully demonstrate #0 and #1** This is an extension of **#2**. It proves full functionality of each function in the LoRa library and makes use of each Grove driver.	Any developer learning to use **#0** or **#1** might use this as reference material. The priority for this is Won't Have because it's impractical for this project's scope.	Functional	User	Won't Have
4	**Develop a demo that demonstrates the advantages of LoRa** This is an extension of **#3**. It would prove that LoRa has the characteristics of an LPWAN[25].	This would provide developers with proof of a motive to use LoRa and consequently **#0**.	Functional	User	Could Have
5	**#0 and #1 will be designed for the UWESense microcontrollers.** The UWESense microcontroller is the main target for this project; **#0** and **#1** will be designed with its specifications in mind.	See **6.2.1** and **6.2.2**	Non-Functional	System	Must Have

[25] See Figure 6.

6	**#0 and #1 will be tested on a UWESense microcontroller.** This is the main target for the project, consequently it will be the board that the LoRa library and Grove module drivers are tested on. It will also be used for the demo in **#3** and **#4**.	See **6.2.1** and **6.2.2**	Non-Functional	System	Must Have
7	**#0 and #1 will be tested on other compatible microcontrollers.** As stated in **6.2.1**, **#0** and **#1** are being designed using generic drivers, therefore they *should* work on compatible microcontrollers. However, without testing this, there's no way of knowing.	**#2, #3, #4** could be demonstrated on several compatible microcontrollers. Successful testing of this would increase the usability of **#0** and **#1**. Its priority is 'Could Have' because this will depend upon time and budget.	Non-Functional	System	Could Have
8	**#0 will only be tested using the 868Mhz (EU) frequency.** See **6.3.2** for details. The only frequencies permitted in the EU are the 433Mhz and 868Mhz; this project is based in UK, so only EU frequencies will be tested.	"The RN2483 is a fully-certified 433/868 MHz module based on wireless LoRa® technology." (Espotel, 2015)	Non-Functional	System	Should Have
9	**#0 and #1 will adhere to the same programming conventions.** There's only one author of the code, so this won't be difficult. It's an important requirement for anybody reading the code though.	"When you agree on coding standards and conventions, you improve the maintainability and readability of your code." (James Shore, 2007)	Non-Functional	System	Must Have
10	**#2 will demonstrate a full IoT solution stack.** This would include passing data from Grove Sensors, to a microcontroller (using **#1**), to a gateway (using **#0**) and then to a webpage via TCP/IP.	This will demonstrate the viability of **#0** and **#1** and provide a more homebrew example of IoT (unlike the Enterprise example in **2.1.1**).	Functional	User	Should Have
11	**#0 and #1 will work independently from each other.** It's likely for these two bits of software will be used in conjunction frequently. Having them require one another would be ridiculously bad design though; they shouldn't inherently require one another.	A developer might be using a different type of Grove module or a Grove module that isn't included in the set of Grove Sensor drivers in **#1**. If **#0** and **#1** depended on each other, this developer would be unable to use **#0**.	Non-Functional	System	Must Have

12	**Git will be used for version management.** Version management is important for any project, it's familiar and universally usage in industry.	There might be a large mistake during development and only find out later. Git allows for rollbacks to a previous version, fix the mistake and merge it with the current development branch.	Non-Functional	System	Must Have
13	**#0 will have its own Git repository.** See **#11**. Developing **#0** with its own Git repository will also allow for modular development.	There might be a mistake during the development of **#0** and need to go back to a previous version. With its own Git repository, no commits made in **#14** will be lost.	Non-Functional	System	Must Have
14	**#1 will have its own Git repository.** See **#11**. Developing **#1** with its own Git repository will also allow for modular development.	If a mistake is made during the development of **#1** and need to go back to a previous version. With its own Git repository, no commits made in **#13** will be lost.	Non-Functional	System	Must Have
15	**Testing and Development will follow the methodology defined in 7.**	(From **7**) "The project will use TDD in combination with a few sprints that will allow me to complete the project in clear milestones."	Non-Functional	System	Should Have
16	**#1 will include a driver for *every* Grove Sensor available.** This would include every available Grove-Sensor as of September 2017.	This is impractical for this projects scope. (From **6.2.3**) "There seems to be a Grove module for every sensor/actuator/display type imaginable"	Functional	User	Won't Have
17	**#1 will include a driver for *every* Grove Actuator available.** This would include every available Grove Sensor as of September 2017.	This is impractical for this projects scope. (From **6.2.3**) "There seems to be a Grove module for every sensor/actuator/display type imaginable".	Functional	User	Won't Have
18	**#1 will include a driver for every sensor found in the Grove Starter Kit[26].** This would be a good expansion of **#1** and sensible middle-ground between **#1** and **#16**.	This would make **#1** a much more viable option for anyone looking to use the kit. A developer looking to start in IoT might buy a Grove Starter Kit and be able to use **#0** and **#1** for both with this requirement met.	Functional	User	Could Have
19	**#1 will include a driver for every actuator found in the Grove Starter Kit.** This would be a good expansion of **#1** and sensible middle-ground between **#1** and **#16**.	This would make **#1** a much more viable option for anyone looking to use the kit. A developer looking to start in IoT might buy a Grove Starter Kit, an Arduino and be able to use **#0** and **#1** for both with this requirement met.	Functional	User	Should Have

[26] https://www.seeedstudio.com/Grove-Starter-Kit-for-Arduino-p-1855.html (01/11/2017)

20	**#0 and #1 will set their settings upon start-up.** This will be the *initialisation* phase of the software. These settings will be run every start up and set the setting that will be required for the software to interact with the hardware properly.	This will improve the fault-tolerance of **#0** and **#1**, ensuring that all settings required for the software to work are set.	Functional	System	Should Have
21	**#0 and #1 will have functions that allow for settings to be changeable during run-time.** These settings will be required for certain use cases and will also increase the flexibility of **#0** and **#1** massively.	A user might use one of these functions to change the data rate being used with **#0** during run-time if a certain condition is triggered.	Functional	System	Must Have
22	**The settings set in #18 will be based upon a configuration file.** A configuration file will improve usability and the ease of testing **#0** and **#1**.	A user might have one configuration file for a certain system using **#0** and another configuration file for another system using **#0**. They'll be able to swap these configuration files without problem.	Functional	System	Should Have
23	**If no configuration file is found (#20), #0 or #1 will use a default configuration.** This will improve the fault-tolerance of **#0** and **#1** if **#20** is included. It will be important to notify the user somehow so they're able to diagnose **#0** and **#1**.	If a user forgets to include a configuration file, **#1** would still run using the default values. The user would be notified through **#23** (if met, if not the process would be documented in **#23, #24** and/or **#25**)	Functional	System	Should Have
24	**#0 and #1 will notify the user of any run-time problems or events.** It's important the systems can tell their users why it's doing what it's doing.	**#1** finds an error during runtime and error exits. The user will be able to catch this error through this notification.	Functional	System	Should Have
25	**The source-code of #0 and #1 will include in-line documentation (comments).** These comments will explain any ambiguous code or decisions made, this should mean any user reading the code won't have trouble understanding it.	A user is trying to understand how part of **#0** works, to do this they'll be able to look at the source-code and follow it; this requirement should make looking at that source-code and following it easy.	Non-Functional	System	Must Have
26	**#0 and #1 will include API documentation.** This would be done using software such as Doxygen. Using software to do this would also enforce **#9**.	A user quickly wants to find out the parameters for a function found in **#0**. Instead of looking through the source code to find this information, they'll simply be able to refer to this as a reference.	Non-Functional	User	Should Have

27	**#0 and #1 will be fully documented.** This includes **#23**, **#24** and some form of user-guide (at least). It might also include other forms of documentation.	Any user looking to use **#0** or **#1** would be able to decide whether it fits their use case and determine how easily they can use it based on this documentation.	Non-Functional	User	Could Have
28	**Each module driver (#1), will work independently of each other.** Doing otherwise would be bad design. It is possible that these will require a shared header or file, but it's important to keep **#1** modular.	It's incredibly unlikely a user will want to use all the sensors at once. Therefore, it makes the most sense to keep the design of **#1** modular.	Functional	System	Should Have
29	**#0 will come under a file size of 1MB.** It's unlikely that the file size of **#0** will go over 1GB, it's important that this requirement is met due to the limited memory of microcontrollers.	A user has a limit of memory space on their microcontroller, using a library that is larger than this size would be non-sensible design decision.	Non-Functional	System	Must Have
30	**#0 will come under a file size of 50kb.** This is an expansion of **#28**. At this stage of the project the final file-size of **#0** can't be determined, this would be a good file size to aim for though.	A user writes a complicated LoRa program for the microcontroller that takes up a lot of space, using a library that takes up minimal space would be an important requirement for them.	Non-Functional	System	Should Have
31	**#1's file-size will come under 1MB.** This is a sensible goal as a file-size for **#1**. The ultimate size will depend upon how many sensors have drivers written for them (see **#17**).	A user wants to use a lot of different sensors and needs to use a lot of the sensor drivers in this project, it's important that there's room left on the microcontroller for their program too.	Non-Functional	System	Should Have
32	**File-size of individual drivers in #1 will be 25kb max.** It's important that the individual drivers in **#1** have a small file-size for the same reasons as **#28**, **#29** and **#30**.	A user writes a complicated program that uses a lot of space on their microcontroller. It's important that any sensor driver they're using doesn't take up too much space.	Non-Functional	System	Should Have
33	**#0 and #1 will only read the configuration file (#21) once. This will be during the *Initialisation* part of the software's runtime.** The other option would be to have them read the configuration files at a set interval, however this would decrease the fault-tolerance of the software and is (arguably) unnecessary as there will be functions that allow the user to change settings during runtime.	A user has default settings in a configuration file that are set during start up (see **#20**). Then they change those settings during run-time (see **#19**). They don't want the settings set during run-time to stay after run-time though.	Functional	System	Should Have

34	**If the configuration file (#20) has syntax errors or invalid values, #0/#1 will notify the user (#23) and *Error Exit*.** This is the safest option in this case. While **#0** or **#1** could simple continue with hard-coded defaults, it would be unintuitive to the user.	A user makes a syntax errors while writing a configuration file for **#0**. **#0** detects this during run-time and notifies the user, then error exits. This tells the user their error and allows them to fix it before the next run-time.	Functional	System	Should Have
35	**Valid values for the config file (#20) will be communicated to the user through documentation.** Not including this would require the user to have expert knowledge of the LoRaWAN specification, this shouldn't be expected of the user.	Somebody new to LoRa wants to use a library for a hackathon and doesn't have time to learn *everything* about LoRa. They should be able to pick up **#0** and use it without spending hours reading about LoRa.	Non-Functional	User	Should Have
36	**#2 will be ready by the Project in Progress Day.** I'd like to show off basic functionality of **#0** and **#1** at this stage (note: they won't be finished).	It will be a good indicator for the progress of the project.	Non-Functional	User	Should Have
37	**#0 and #1 will be written in C/C++.** As noted in **6.2.4**, C/C++ are the most popular choices for Embedded.	See (ISO/IEC, 2006).	Functional	System	Must Have
38	**#0 will be compliant with the LoRaWAN specification** (N. Sornin (Semtech), 2015) The LoRaWAN specification outlines the operation of a LoRa device. It's intended to be applicable to both private and public networks.	While the RN2483 Module does have the capability to function outside the limits of the LoRaWAN specification, this library will only make use of the LoRaWAN (Class A) compliant capabilities so that the user isn't able operate outside of them and cause themselves unintended issues.	Non-Functional	System	Must Have
39	**#0's functionality of joining a LoRa network will not be blocking** While making it blocking would be more fail-safe for the user, it would also limit the use ability of the library.	If the user wanted to implement a callback function that's run after the Join has complete, a blocking function would stop this.	Functional	System	Must Have
40	**#0 will be written in C** Using C++ over C would provide little benefit in this case, since the library will be simple	This should improve the compatibility of the library, since C is still more popular than C++ in embedded (see **6.2.3**)	Functional	System	Must Have
41	**#1 will be written in C++** This is an instance where C++ provides a lot of benefit through its OO paradigm	This will make development of the drivers much easier without introducing any overhead (see **6.2.4**)	Functional	System	Must Have

Table 3. Project Requirements Table

9 DESIGN

9.1 LoRa RN2483 Library

These diagrams describe the functionality of the LoRa RN2483 library. As stated in **requirement 38**, the functionality of the library will be limited to what the LoRaWAN specification allows. It's also important to know that these diagrams do not fully illustrate the error handling of the library (e.g. if user passes an invalid parameter). In this case the library will notify the user via **requirement 24** and handle the error appropriately (most likely passing it to the user).

9.1.1 Use Cases

- The role of Developer could encompass different LoRa devices using the RN2483 Library (e.g. a LoRa node or single-channel LoRa gateway).

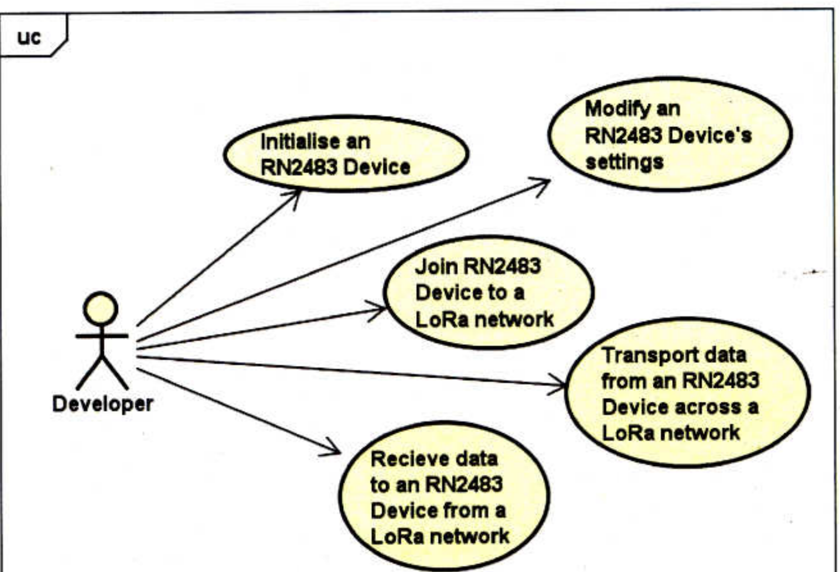
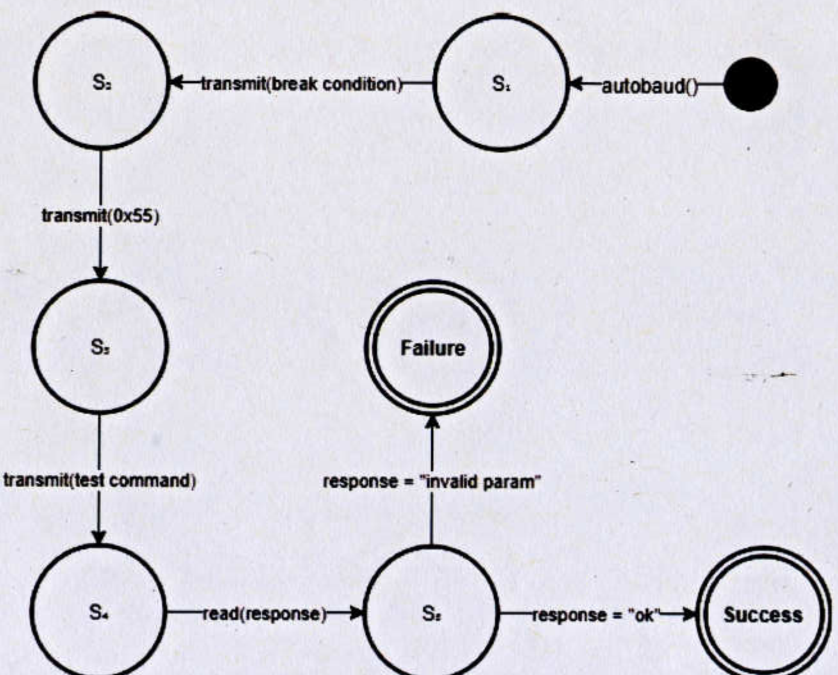


Figure 16. LoRa RN2483 Library use cases

9.1.1.1 *Use Case – Initialise an RN2483 Device*

- For a device to tx/rx over a LoRaWAN network, the settings in initMAC are *required* to be set. This function is designed to set them all for the user at once.
- Within the *initMAC()* function, the parameters of the settings (e.g. device address) are read from the 'Configuration file' (**requirement 22**).
- The *autobaud()* function is optional and in some cases might not be required, however it's recommended to ensure successful communication.
- The *initMAC()* function could be replicated manually by using the RN2483 library calls to manually set each of these settings.

Figure 17. autobaud() State Machine



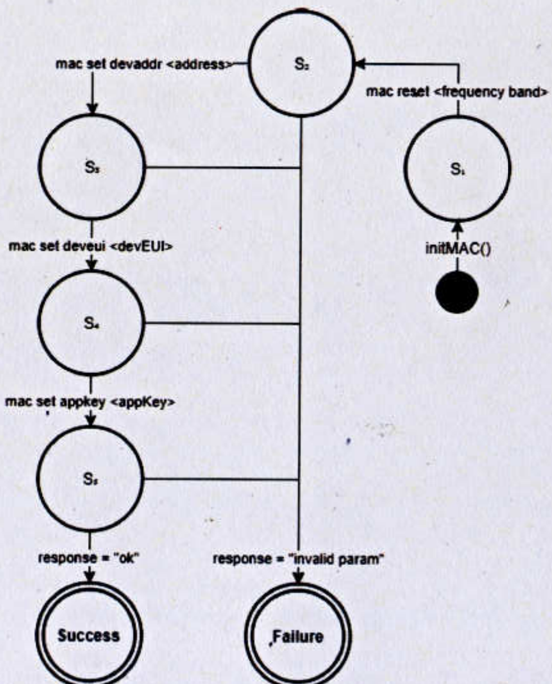


Figure 18. initMAC() State Machine

9.1.1.2 *Use Case – Join RN2483 Device to a LoRa network*

- In most cases, the *Microcontroller: Developer* should expect to wait for a few seconds to allow the network to respond after the RN2483 Library has confirmed the join request has been sent (see **requirement 39**).

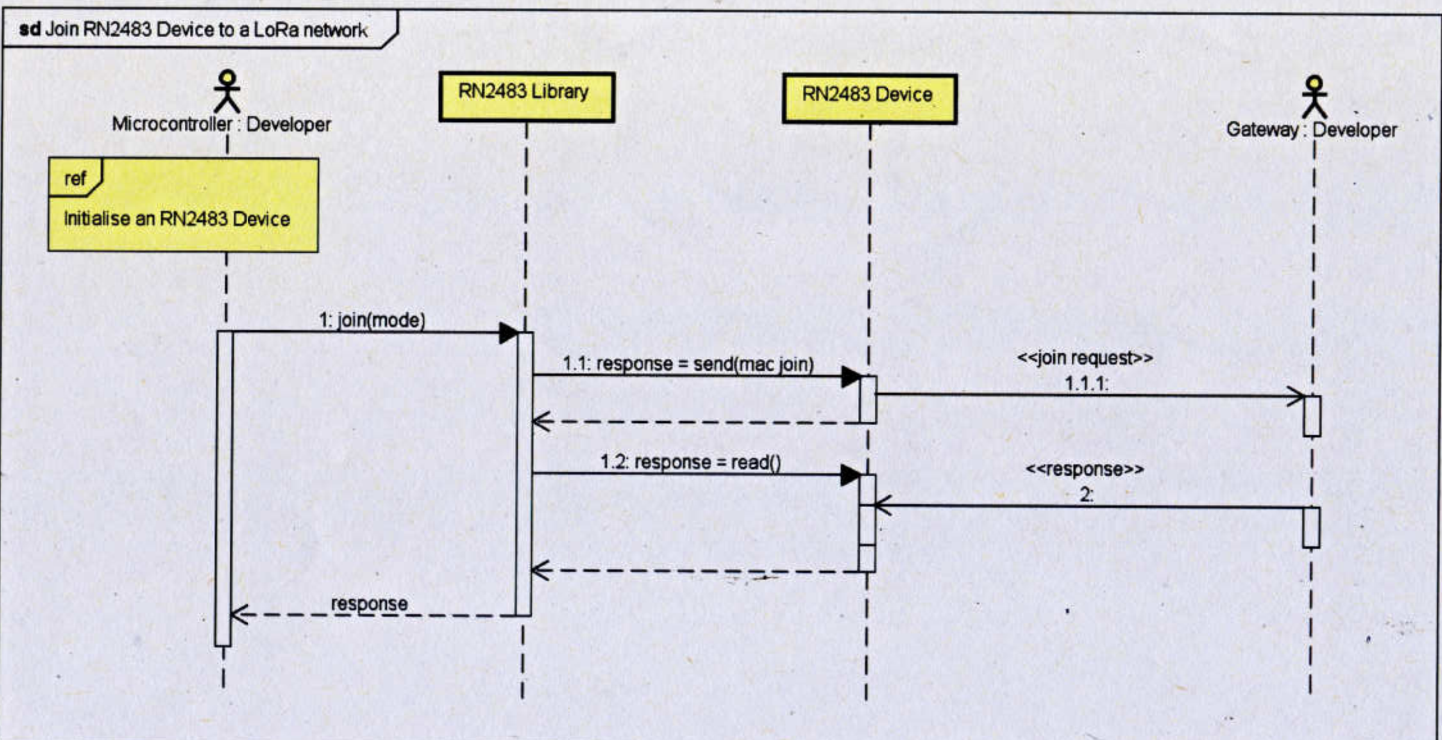


Figure 19. RN2483 Join procedure

9.1.2 Use Case – Send/Receive data over LoRa

- The RN2483 Device does have an option continuously receive data. Unfortunately, this option isn't compliant with the LoRaWAN specification, not only this but due to the limitations of LoRa it's likely to cause undefined behaviour. See **requirement 38**.
- Since the RN2483 is a Class A device, the only way for it to receive data will be in its two receive windows (see **Figure 12**).
- The option for the user to confirm their transmission decides whether the library will expect to receive an ACK from the server.

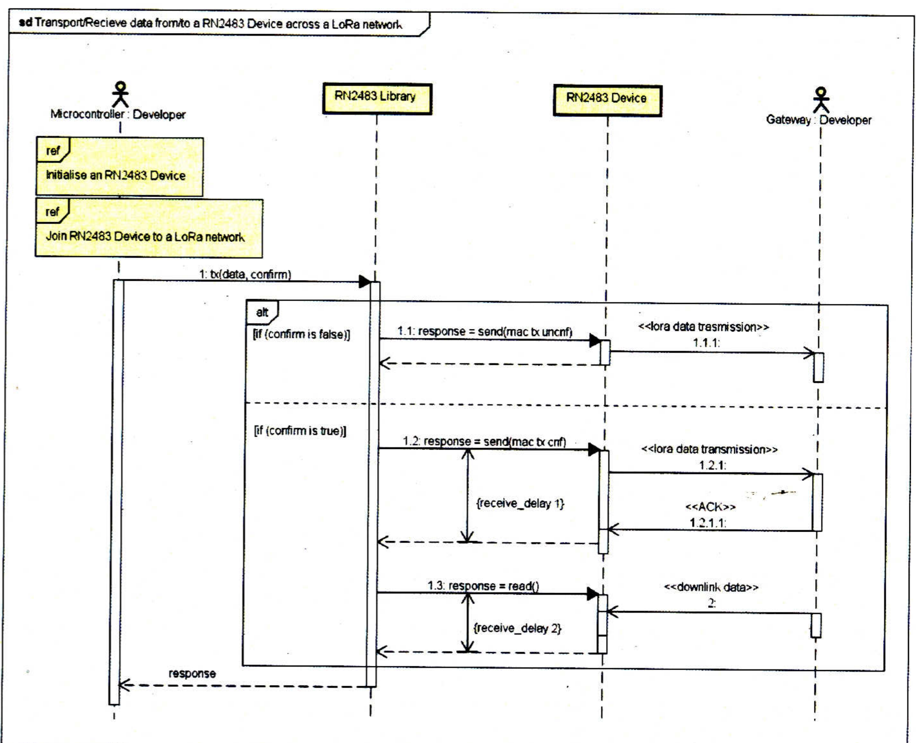


Figure 20. Send/Recieve data over LoRa

9.2 GROVE DRIVERS

This is a class diagram to specify the design of the framework for Grove Drivers. Since all the Grove Modules are standardized (see **2.2.2**), it was difficult to figure out how to differentiate them into a way that would make sense for the design. Small intricacies between modules made having a single *GroveModule* base class infeasible.

Luckily seeed have already categorised the modules on their Wikipedia. 'Sensor', 'Actuator', 'Display', 'Communication' and 'Other'. While the project only tests the design's suitability with Sensors and Actuators, it allowed proved its flexibility by allowing me to add a Communication base class and a Communication module with relative ease.

Below is the final design. This design evolved through the project, see **Appendices, Previous Grove Designs**, there are notes on this in **7.1.1/S0-R0** too.

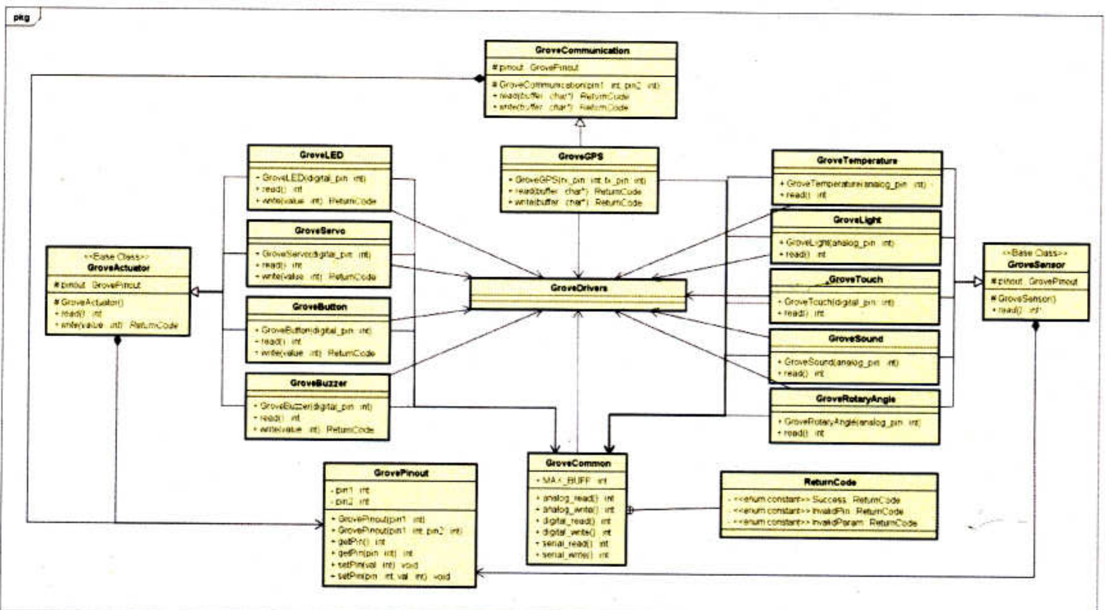


Figure 21. Grove Framework design (final implementation)

The *GroveGPS* class was a late addition due to a change in what my demo would use for its Grove Module (see 6.3/S3-R1) and its addition managed to prove the flexibility of the design to be successful.

The *GroveCommon* class is where any hardware specific functionality required (mostly tx/rx) will be contained, so that porting the framework to various platforms would be simple. This was also a success, since all its functions are just wrappers that that the modules call upon.

When porting to the BBC Micro:Bit, this was the only file that required modifying.

9.3 DEMO

This flowchart illustrates the control flow of the main function running on the BBC Micro:Bit in the demonstration of the Grove Framework and RN2483 Library. Initially the demo was using the GroveTemperature and GroveButton modules. After review (see **7.1.3/S3-R1**), it made more sense to use a Grove GPS module to demonstrate the range of LoRa. So, I simply swapped out the Temperature and Button modules for the newly added GPS module.

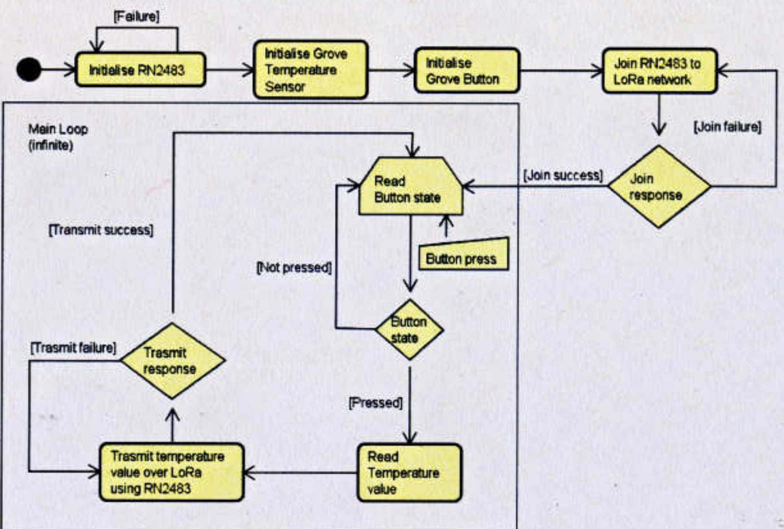


Figure 22. Initial demo design

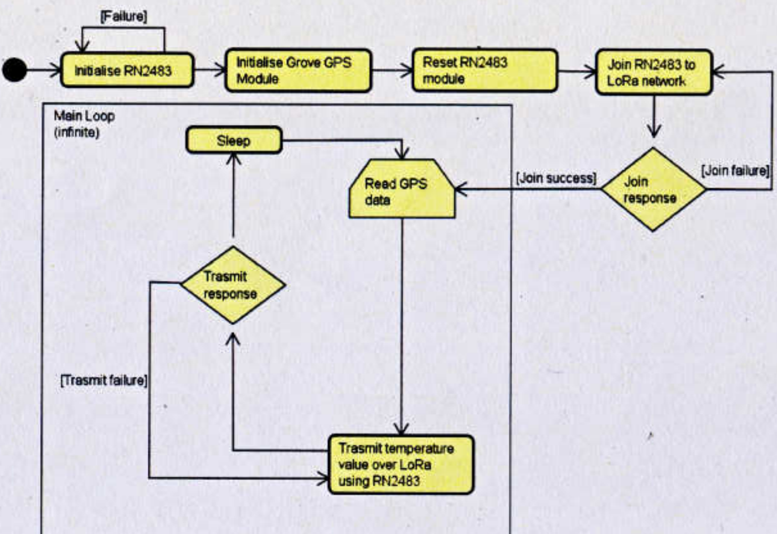


Figure 23. Final Demo design flowchart

10 Planning & Testing

10.1 Sprints

ID	Sprint Goal	Start Date	End Date	Post-Sprint notes
S0	Prototype the design in **7.2**. Aim to write a driver for each module in the Grove Starter Kit (see **requirements 18 & 19**).	16th October 2017	22nd October 2017	I was only able to write drivers for: Temperature, Light, Touch sensors and the Button actuator. This helped me flesh out and figure out how to improve the design in **9.2** a lot (see review **S0-R0**)
S1	Establish tx/rx communication with a Microchip RN2483 module and test various functionalities.	6th November 2017	12th November 2017	I was able to successfully tx/rx using a LoRaBee RN2483[27] attached via cables.
S2	Join a Microchip RN2483 to a LoRa network and tx some dummy data to it	15th November 2017	21nd November 2017	Unfortunately, a LoRa gateway couldn't be sourced and there was no public TTN gateway in range, meaning that tx'in data over a LoRa network wasn't possible.
S3	Create a prototype of the demo in **7.3**	7th December 2017	13th December 2017	Due to the issues in **S2**, I couldn't TX any data. Instead I did manage to write a program that *should* function correctly (provided the join and tx from **S2** work when resolved).
S4	Establish tx/rx communication with the UWESense MCU and the on-board RN2483 module	15th January 2018	21st January 2018	The UWESense uses the STM32 HAL (which I found challenging), however I managed to successfully flash the board by loading a compiled '.hex' onto it through the ARM toolchain's GDB and monitor it via OpenOCD. I also implemented a function that write to the OpenOCD instance via SVC calls (which required assembly). I also managed to reset the RN2483 module with the RESET pin but am currently unable to parse the firmware version it sends back. I haven't been able to successfully send out any commands that prompt a response.
S5	Communicate to Grove Modules on the UWESense MCU using its on-board Grove ports	-	-	Since RN2483 communication hasn't been successful, the target MCU has had to change to a BBC Micro:bit. See **11.2.1** for details.
S6	Create the demo (design **9.3**)	26th March 2018	1st April 2018	Development on this got pushed back due to issues with the UWESense. After finishing **S7** and **S8**, I was able to go ahead and develop it. I did run into issues unfortunately, this is due to previous problems – these are covered in **S6-R0**.

[27] An RN2483 Module soldered to an XBee breakout board.

S7	Write the library for the RN2483 through the BBC Micro:bit on the breakout board	12th March 2018	18th March 2018	This was frustrating. I implemented all the basic logic in a main branch with the work I'd done while trying to communicate via the UWESense and wrote hardware specific functions using stdlib i/o and then created a derived git branch with those functions bespoke for & tested on the BBC Micro:Bit.
S8	Write the framework for Grove Module Drivers on the BBC micro:bit through the breakout board's Grove ports	19th March 2018	25th March 2018	I was successful in doing this. Due to time restraint I had to test it as I went, but managed to create a non-platform specific version from all the pre-work I'd done before the sprint while trying to get the UWESense to work and then a git branch derived from that designed specifically to support the BBC Micro:Bit.

Table 4. Sprint Schedule

10.2 Testing

ID	Test Case	Priority	Result	Notes
S0-T0	Compiles without errors	Must	✓	I was using the Arduino platform, so the specifics of the Makefile were obscured from me.
S0-T1	Compiles without warnings	Should	✓	I was using the Arduino platform, so the specifics of the Makefile were obscured from me.
S0-T2	Values can be correctly written to/read from each module that a driver that has a prototype written for it	Must	✓	Correct values were read to/written from each driver that was written.
S0-T3	Find flaws in the design in **7.2**	Should	✓	See review **S0-R0**
S0-T4	The prototype software runs on multiple embedded platforms	Could	✗	Unfortunately, because I was using the Arduino platform, the Arduino library (not portable) was required to communicate with the hardware. The logic shouldn't need to change for other platforms though.
S1-T0	Get a firmware version	Must	✓	I achieved this though **S1-T2**
S1-T1	Get the RN2483 modules hardware/device eui using the *"sys get hweui\r\n"* command	Must	✓	This was done to verify the tx/rx and make sure the version was being retrieved from something like turning the RN2483.
S1-T2	Toggle the reset pin on the RN2483 module	Should	✓	I set RN2483's reset pin to LOW and then back HIGH again. This causes the RN2483 to put the firmware version into the buffer.
S1-T3	Activate the autobaud functionality of the RN2483	Should	✗	I did write code to do this, however I wasn't able to convincingly test it since I wasn't sure how to change baud rate during runtime on an Arduino. The RN2483'd default baud rate of 57600 was worked for tx/rx though.

S1-T4	Pass **S1-T0**, **S1-T1**, **S1-T2**, **S1-T3** on multiple embedded platforms	Could	✓	I couldn't achieve **S1-T3**, however I was able to achieve the rest with an Intel Quark (had one lying around and I was familiar with its HAL).
S2-T1	Join a public LoRaWAN network (or my own)	Must	✗	See **S2** Sprint Notes.
S2-T2	Tx dummy data over a joined LoRaWAN network	Must	✗	See **S2** Sprint Notes. Without joining a public LoRaWAN network or having a local LoRa gateway, there was no way of testing whether the data was tx'ing correctly.
S2-T3	Rx data via a LoRaWAN network using the downlink capability	Should	✗	See **S2** Sprint Notes.
S3-T1	Correctly receive data from a Grove Temperature Sensor	Must	✓	Since I was still using an Arduino as my prototyping platform, I simply took the work from **S0** without issues.
S3-T2	Only receive sensor data when Grove Button is pressed	Must	✓	Since I was still using an Arduino as my prototyping platform, I simply took the work from **S0** without issues.
S3-T3	Successfully tx data received from a Grove Temperature Sensor over a LoRaWAN network (when button is pressed)	Must	✗	See **S2** Sprint Notes.
S3-T4	Successfully receive tx'd data on the application	Must	✗	See **S2** Sprint Notes
S4-T1	Successfully flash the UWESense MCU board	Must	✓	This was done through the ARM toolchain's GDB to load the compiled '.hex' onto it and an STM32 Programmer with OpenOCD.
S4-T2	Confirm the flashed software is running on the MCU board	Must	✓	I did this by toggling one of the LED's in a loop.
S4-T3	Get a firmware version from the on-board RN2483.	Must	✓	I think I've achieved this through the UWESense the same way I did it on the Arduino (using the RESET pin). Unfortunately, the data in the RX buffer is printing to (via the SVC function) as garbled data. I've not figured out if this is a baud rate issue or a memory issue yet (it's more challenging due to the STM32's HAL). See **S4-R*** for notes & updates.
S4-T4	Get the RN2483 modules hardware/device eui using the *"sys get hweui\r\n"* command	Must	✗	When I tried transmitting the command over the UWESense using the STM32 HAL the RX buffer stayed empty. See **S4-R*** for notes & updates.
S4-T5	Toggle the reset pin on the RN2483 module	Should	✓	I set RN2483's reset pin to LOW and then back HIGH again. This causes the RN2483 to put *something* into the RX buffer, however I've not been able to read it properly
S4-T6	Activate the autobaud functionality of the RN2483	Should	✗	I did write code to do this, however I wasn't able to convincingly test it since I've not been able to correctly receive data (even on the default baud rate).

S7-T1	Compiles without errors	Must	✓	In the general-platform master branch I've got a Makefile that produces a static library (.a) file, the microbit version required yotta integration
S7-T2	Compiles without warnings	Should	✓	The general-platform master branch did. The microbit version doesn't have any warnings that are specific to my library but does have a lot that are a consequence of the DAL & yotta build process
S7-T3	Commands can be sent to the RN2483 and the user is able to view the response	Must	✓	The *RN2483_command* function does this, and writes the response to a user-defined buffer
S7-T4	Able to reset the RN2483	Must	✓	This was only implemented on the microbit branch since it's platform specific (pin toggling).
S7-T5	The initMAC function stops at first error	Must	✓	I tested this by setting invalid values in the cfg file
S7-T6	The library is able to read settings from the configuration file (see **requirement 22**)	Should	✓	I used a Makefile to write the values from the .cfg file to a header which defines a set of macros called on by the library
S7-T7	The initMAC function can set all settings when all values are valid	Should	✓	It takes a while to run since because of the time it takes for the RN2483 to responds
S7-T8	The join function is able successfully join a LoRaWAN network	Must	✓	I tested this using The Things Network and local gateway
S7-T9	The join function can report to the user reasons for failure	Should	✓	This is done by reading the response from the RN2483 and returning one of the Return Codes which are documented for the user
S7-T10	The join function can join using ABP and OTAA	Should	?	The command successfully goes to the RN2483, however I wasn't able to join a device to TTN using OTAA. I don't know if this is because of TTN or the RN2483 module, however it does respond as it should to a failure.
S7-T11	The TX function can transmit a LoRaWAN packet	Must	✓	This was tested using TTN and a local gateway
S7-T12	The TX function can report reasons for failure	Must	✓	This is done by reading the response from the RN2483 and returning one of the Return Codes which are documented for the user
S7-T13	The TX function returns any downlink data to the user	Should	✓	I was able to successfully do this. I also implemented functionality to convert the hexadecimal string received to ascii.
S7-T14	The library can run without causing a crash on the platform	Should	✓	It didn't initially, however I investigated the issue and found it was caused by memory space. I reduced the memory used by the library by as much as possible and fixed this.

S7-T15	The library is easy to port between platforms	Could	✓	Due to its build process the microbit was more difficult to port to than I imagine most other platforms would be. Ideally, you should only need to change the wrapper read/write functions though.
S8-T1	The framework compiles without errors	Must	✓	For the *master* branch I didn't have way of testing this, but the derived microbit branch did without many changes.
S8-T2	The framework compiles without warnings	Should	✓	The yotta build process tends to obscure a lot of this, but I couldn't see any related to the framework
S8-T3	All the drivers in the framework return the correct data when read from	Must	✓	I tested this and confirmed the results by testing it with results from proven code on the Arduino platform
S8-T4	All the drivers (that allow for it) can be successfully written to	Must	✓	The only driver that I ended up writing which made use of being written to, was the buzzer (It made a loud noise when I did (as it should)).
S8-T5	All the drivers are easy to implement into a program	Should	✓	While I didn't write a factory pattern in the end, I believe it was made simpler by doing so
S8-T6	The framework is easy to port to different platforms	Could	✓	I managed to keep all platform-specific functionality in a single file that contains wrapper functions for things like serial r/w. When porting it to the microbit, the was the only file I had to change.
S6-T0	Read data from a selected Grove Module correctly using the Grove Framework	Must	✓	See **S8-T3**
S6-T1	Successfully TX data over LoRa using the RN2483 library	Must	✓	See **S7-T11**
S6-T2	Successfully RX data over LoRa using the RN2483 library	Could	✓	See **S7-T13**
S6-T3	The data can be read on the server un-malformed	Must	✓	Data integrity is important, and I couldn't find any issues with it in this demo.
S7-T4	The data sent over LoRa is displayed in some form to users	Should	✗	I'm able to do a *curl* to fetch the data, however I didn't have time to write a webpage that could make a cross-site request for the data in time.

Table 5. Test Results

11 Implementation

11.1 Stage 1 – Prototyping

This initial stage was for prototyping. This was in-part because the target platform for the project wasn't ready yet, but also to test the initial designs in **9**. The prototyping was done on an Arduino UNO.

This stage of the project consisted of 4 sprints (**S0, S1, S2, S3**).

11.1.1 Grove Modules

11.1.1.1 Overview

The prototype of the Grove Framework was a mixed bag. Implementing the initial design (**15.2, Figure 27**) demonstrated a lot of the flaws that I hadn't previously considered.

As well as the improvements noted in **S0-R0**, the main issue was with the Factory Class in GroveDrivers – it proved redundant. The aim of the factory class was to provide the user with a simple interface to the classes without them needing to memorize the layout of the library, this was just as easily achieved using a sensible naming scheme for the classes and making sure that they all adhere to the same standard interface[28].

The standard interface will be documented through the Doxygen documentation (**requirement 26**), and the classes will be called directly using the standard naming convention. See **Figure 30**.

11.1.1.2 Sprint Logs

11.1.1.2.1 S0-R0, 25th October 2017

This sprint gave me a good feel for the flaws of and improvements to be made to the design in **Figure 27**. It demonstrated that more than one abstract base class would be required for the Grove Modules. Luckily seeed categorise their modules on their wiki (seeed studio, n.d.), and these fit perfectly into a set of base classes that would accommodate all the intricacies of separate Grove Modules.

I also decided that it would be necessary to add a class to handle pin logic to avoid repeating get/set functions of pins. The changes to the design are reflected in **Figure 28**.

11.1.2 RN2483

11.1.2.1 Overview

Implementing the prototype of the RN2483 was simple because of the Arduino platform's reliable serial communication. This sprint was useful for testing the logic in **9.1** and familiarising myself with

[28] Refers to the classes having a standard set of functions that the user can rely on being implemented.

communicating with the RN2483. While the serial tx/rx will require re-implementing in the final implementation, the rest of the code written should be portable.

The aim of the **S2** was to test actual LoRa communication using the RN2483. As noted in **10.1/S2**, this sprint didn't prove to be a success, as I wasn't able to get a LoRa gateway and wasn't in range of a public network gateway – this gave me no option to send packets over a LoRaWAN network. It also meant that the sprint and all the test cases had to be carried over until I had the means to achieve them (*this issue was resolved in **11.2.3***).

11.1.2.2 Sprint Logs

11.1.2.2.1 S1-R0, 13th November 2017

I didn't have any problems or need to make any improvements to the design in **9.1** during/after this sprint. As noted in **S1-T3** I couldn't implement the *autobaud* functionality in time and wasn't familiar enough with the Arduino UNO platform to know how to change the baud rate of the serial communication during runtime. Since this was only a prototype, and mainly focused on testing design and familiarising myself with the project hardware[29], I didn't want to waste time figuring this out.

11.1.2.2.2 S2-R0, 22nd November 2017

I didn't have the budget for a Semtech SX1301[30], so I tried to setup a budget, single-channel gateway using a Raspberry Pi and an RFM95 (using various repositories and following various guides) but had no success. The other solution to this problem was to join using a public LoRaWAN gateway, however this wasn't possible since there wasn't a gateway in range of my location.

11.1.2.2.3 S2-R1, 08th February 2018

I've been able to borrow a LoRa gateway and successfully register it on the TTN. Unfortunately, I'm unable to communicate with the RN2483 module on the UWESense currently (*see **11.2.1***), so I won't be able to resolve these test cases until those issues are resolved. While I could use the prototype from **S1-R0**, doing so would take time and I'm currently focused on the UWESense issue.

11.1.3 Demo

11.1.3.1 Overview

Prototyping the demo was mostly a failure due to the issues from **S2**. This meant I couldn't test the full demo and could only prototype the software required on the embedded device, which had already been proven in **11.1.1** and **11.1.2**.

---

[29] This doesn't include the Arduino platform.

[30] The multi-channel Semtech LoRa chip used in standard LoRa gateways (see **6.3.2**). They're usually prices around £300.

11.1.3.2 Sprint Logs

11.1.3.2.1 S3-R0, 16th December 2017

The issues from S2 are have carried over to cause issues with this sprint. I was able to write everything on the embedded-end successfully; once S2's issues are resolved, I'll be able to test it.

11.1.3.2.2 S3-R1, 31st January 2018

After a review of the project, the design in **9.3** was amended from **Figure 22**, to **Figure 23**. The Grove Temperature sensor was swapped out for a Grove GPS communication module, with the intention to test to capable ranges of LoRa. Doing so has allowed me to test the flexibility of the design for the Grove Framework in **9.2**, since I was able add the *GroveCommunication* base class and *GroveGPS* module which inherits from it, without any problems.

11.2 Stage 2 – Platform Setup

The purposes of this stage were to familiarise myself with & test the platform that I'd be developing for. This is an important stage in any embedded project since each platform has its own intricacies which require learning and possibly overcoming before *real* development can start. Skipping this stage could cause serious issues later and possibly result in the project ending half-finished and/or broken. Only 2 sprints were needed for this stage (**S4, S5**).

11.2.1 UWESense

11.2.1.1 Overview

The UWESense proved to be a frustrating experience. While most MCU's require their own domain knowledge, generally after working with them for a while the developer's able to get to grips with its functionality. I achieved this and learnt what functionality in the STM32 HAL I would need to use in my project, as well as being able to add functionality, such as SVC functions in Assembly.

During this stage, I realised that I wouldn't be able use a configuration file as intended in **requirement 22** since it's rare for an embedded platform to have a file system. Instead I decided it would be best to using a header file to set the configuration options using *#define* macros.

Unfortunately, I wasn't able resolve an issue that stopped me being able to reliably communicate to the RN2483 module. Consequently, I've had to move development to targeting the BBC Micro:Bit. See **11.2.2** for details.

11.2.1.2 Sprint Logs

11.2.1.2.1 S4-R0, 28th January 2018

This was a lot more challenging than I was expecting it to be. While moving to new embedded hardware is usually a learning curve, I found the STM32 HAL unintuitive. Despite this, I was able to pass all the tests specifically related to the UWESense board. I've also been able to write a few SVC functions (which required Assembly, see **6.2.4.1**) – specifically, one that allows me to print to the GDB debug session, which will help *massively*.

Unfortunately, I've not been able to RX serial data correctly from the RN2483 module. I'm able to get (what I assume is) the version number via the RESET pin, printing it via the SVC function displays a garbled string though.

11.2.1.2.2 S4-R1, 15th January 2018

I've been able to resolve the issue of data in the RX buffer being garbled. I was forgetting to add a terminator to the end of the data received when printing it – this is an example of how C allows you to shoot yourself in the foot (**6.2.4.1**).

Unfortunately, I've not been able to make progress in figuring out why serial communication to the RN2483 is failing. I've even tried using a Saleae Logic Analyser to check that the commands are being sent correctly over the serial line.

11.2.1.2.3 S4-R2, 9th March 2018

Even after reviewing the issues of serial communication to/from the RN2483 on the UWESense board with my tutor, nobody has been able to figure out the problem or a resolution. This likely means the problem is to do with something outside of my control. Since there's currently only a single UWESense board, there's no option to test if it's a hardware issue aside from already tried methods.

Consequently, I've made the decision to change the Microcontroller being used to a BBC Micro:Bit

11.2.2 BBC Micro:Bit

Due to the problems I faced with the UWESense Board, I've decided that I had to move the target platform to a BBC Micro:Bit to complete the project in time. This certainly isn't ideal for the project, since it means I've had to move to a target that I haven't been able to do as much in-depth research on.

Part of the reason I chose the BBC Micro:Bit is because it's a simple MCU, with basic features. It's also designed to be used as an education tool, while also offering low-level access to the Mbed OS running on it. The idea being that the platform wouldn't provide me with any more problems but would also grant me low-enough level access that I could continue development.

The main reason the I decided to choose the BBC Micro:Bit is because I was able to get a hold of a shield for it with an RN2483 module onboard *and* with two Grove Ports. Obviously, this shield was ideal for my project and would make implementation much easier.

Figure 24. BBC Micro:Bit RN2483 shield

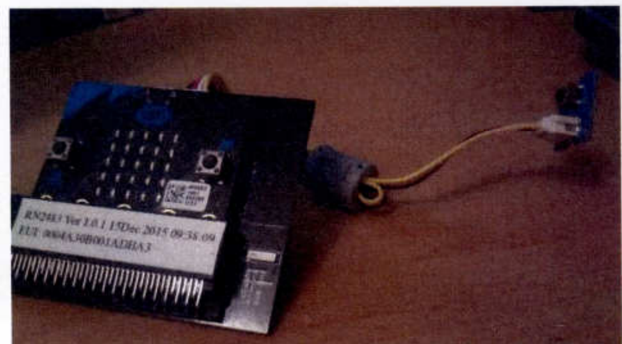


Figure 25. BBC Micro:Bit RN2483 shield (connected)

11.2.3 The Things Network (LoRaWAN network)

The project is going to use The Things Network as the LoRaWAN network that I'll connect to in the demo. This is because it's the only LoRaWAN network in the UK and consequently the only network

I'll have access to. Whilst trying to resolve issues on the UWESense platform, I was able to get a hold of a TTN gateway and resolve issues in **S2**.

11.3 Stage 3 – Final Implementation

Due to the issues on and the prolonged period spent resolving issues regarding the UWESense, this stage was delayed and (in the end) required me to adapt to a new target. Since I was spending so long on **Stage 2**, I'd started doing pre-work for the Grove Framework and RN2483 Library.

I started by moving any portable code from the prototypes to the final implementation, then worked around those elements. Since I didn't have a target platform sorted, any hardware-specific functionality was modularised for later implementation. During this time, the best testing I could do was to make sure there were no compile errors or warnings.

This decision forced me to write the library and framework to be (mostly) hardware independent. Since all hardware specific functionality was being modularised, porting the two programs became much easier (changes only had to be made in one place). This has been reflected in the git used for version management: the *master* branch is a non-platform-specific implementation, using *#error* macros where hardware-specific implementation is required[31]. The repository has individual *platform/* branches which have the hardware-specific elements implemented.

As I was familiar with the BBC Micro:Bit, porting the code wasn't too much trouble. Although the lack of complexity in its components certainly caused problems during **S6**. The main issue was related to the BBC Micro:Bit's limited serial capability.

This stage covers **S6**, **S7**, and **S8**[32].

11.3.1 RN2483 Library

11.3.1.1 Overview

11.3.1.1.1 Configuration file

As stated in **11.2.1**, I decided to use a header file with *#define* macros to configure the LoRaWAN options for *initMAC()* to read. This feature was improved by creating a *.cfg*, who's values get written to the header file during the build process – using a *.cfg* file communicates configurability better to the user.

I did this by writing a Makefile which was used to compile the library into a 'static library file'[33] (before I was able to compile to the target platform[34]) - see *Makefile* in the *master* branch of the RN2483 repository. While the intention was that this build process would be similar in all the *platform/*

---

[31] *#error* macros throw compile errors followed by an error message

[32] Due to the issues in **11.2**, they were done out of order.

[33] Static *'.a'* files are compiled libraries for use in other programs (different from shared *'.so'* libraries).

[34] **(11.2)** The issues with the UWESense needed to be resolved before I compiled the library code for it.

branches, it didn't end up being used for the Micro:Bit due to its awkward CMake build process (which makes adding external files very difficult)[35].

Within the *initMAC()* function, I added *#ifndef* checks to avoid any runtime errors if there was an issue in writing settings the configuration header (see **Figure 26**); this also means that if the setting isn't defined, then the *initMAC* function will skip over it – this might cause issues for the user later on if it's a required LoRaWAN setting.

```
#ifdef LoRaWAN_DevEUI
    ret = RN2483_command(serial, "mac set deveui " LoRaWAN_DevEUI "\r\n", response);
#endif
```

Figure 26. #ifndef guards

11.3.1.1.2 Internal/External declarations

To make sure that no functions or global variables in the library wouldn't override anything in a user's program, every *external* variable/function of the library is prepended with *RN2483_*. This should also communicate to anybody reading code the functions being called from the RN2483 library.

Functionality of the library was split in two: the first was half was of *external* functions, that would be available to the user, these functions/variables are declared in the header file for the user to view. The second half was of *static* functions that only need to be called upon by the library, these functions/variables are only written in the source file and mostly consist of helper and utility functions the library uses.

11.3.1.1.3 Hardware Abstraction

As noted in **11.3**, I/O to the RN2483 was modularised. I did this by creating to two *static* functions: *read()* and *write(uint8_t *string)*. In the *master* branch, the two functions simply contain *#error* macros telling the user they require implementing (see **Figure 27**). On the *platform/mbit* branch, they contain implementation specific to the Micro:Bit (see **Figure 28**).

These *should* be the only two functions that need changing when porting from the *master* branch, however the MicroBit platform required me to pass around it's *Serial* object, which meant a lot more had changes had to be made. I'm speculating the MicroBit is a special case though.

[35] It turned out to be much easier to just copy the header and source files to the build directory of the Micro:Bit yotta project for the demo.

```
20 static uint8_t read()
21 {
22     #ifndef DEBUG
23         #error "This function is platform-specific and requires implementing"
24     #endif
25 }
26 //! write a string of bytes via UART to the RN2483
27 static void write(uint8_t *string)
28 {
29     #ifndef DEBUG
30         #error "This function is platform-specific and requires implementing"
31     #endif
32 }
```

Figure 27. RN2483 master branch I/O

```
19 //! read a single byte from via UART from the RN2483
20 static uint8_t read(MicroBitSerial *serial)
21 {
22     return serial->read(ASYNC);
23 }
24 //! write a string of bytes via UART to the RN2483
25 static void write(MicroBitSerial *serial, const char *string)
26 {
27     serial->printf(string);
28 }
```

Figure 28. RN2483 platform/mbit branch I/O

11.3.1.1.4 Data handling

Two static functions that I had to implement were the *get_hex_string* and *get_text_string* functions. This is because the RN2483 module expects the hex representation of TX data in an ascii string.

The *get_hex_string* (**Figure 29**) is pretty cool – it works by iterating twice over each character in *buff*, in the first iteration (where *i* is even) it pushes the character 4 bits to the left and *AND*'s it with 0x0F – this returns the 'left' hex nibble of a byte. In the second iteration, it simply *AND*'s the character with 0x0F – this returns the 'right' hex nibble of the byte. At the end of each iteration, the nibble returned is used as the index of a lookup table containing the corresponding ascii characters[36].

```
29 // Converts buff into a string representation of it hexadecimal representation
30 static void get_hex_string(uint8_t *buff, int buff_len, char *ret)
31 {
32     int j;  //index of buff
33     int i;  //index of string
34     uint8_t nibble;
35     const char hex_map[16] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
36 
37     for (i = 0, j = 0; i < buff_len*2; i++)
38     {
39         nibble = 20;    // out of bounds of hex_map
40 
41         if (i % 2 == 0)
42             nibble = (buff[j] >> 4) & 0x0F;
43         else
44         {
45             nibble = buff[j] & 0x0F;
46             j++;
47         }
48 
49         ret[i] = hex_map[nibble];
50     }
51 
52     ret[i] = '\0';
53     return;
54 }
```

Figure 29. data to its hex representation in ascii

The *get_text_string* (**Figure 30**) was much simple than the previous. It simply had to fetch two hex character each iteration from the hex string (*hex*) and use *strtol* to convert it to base16.

[36] A 'nibble' is 4 bits (0x00 – 0x0F), which in decimal covers is 0 to 15.

```
55 // Converts string a string representation of hex characters to ascii
56 static void get_text_string(const char *hex, int hex_len, char *ret)
57 {
58     char byte[3];
59
60     for (int i = 0; i < hex_len; i+=2)
61     {
62         byte[0] = hex[i];
63         byte[1] = hex[i+1];
64         byte[2] = '\0';
65
66         ret[i/2] = (char)strtol(byte, NULL, 16);
67     }
68
69     return;
70 }
```

Figure 30. Hex representation of data (in ascii) to a string

11.3.1.1.5 Memory handling

I made the decision not to use the *malloc()* function or any other direct memory allocation/deallocation C functionality. This was done to avoid memory fragmentation and memory leaks within the library, see (Murphy, 2016). Instead, I stuck to using arrays and pointers so that whatever the system the user's platform is running, it would be allocate memory itself. In embedded development, it turns out that this is a much safer option (Murphy, 2016).

11.3.1.1.6 etc

The rest of the library simply follows the designs from **9.1** as you'd expected.

11.3.1.1.7 Examples

As noted in **11.3.3**, I ended up with a demo of just the RN2482 library that tx'd dummy data on the Micro:Bit (using the files from the *platform/mbit* branch). I decided to include this demo in an examples folder of the repository branch (see *examples/* on the *platform/mbit* branch of the RN2483 repository).

11.3.1.2 Sprint logs

11.3.1.2.1 S7-R0, 17th March 2018

This was a lot more frustrating than I expected due to the platform I was now writing for (BBC Micro:Bit). It only allows for a single Serial instance, so to print responses from the RN2483 in a debug console, I had to store data received from the RN2483, redirect the Serial, and then print it to the debug console.

The Micro:Bit's implementation of Serial also proved unstable when swapping pins and baud rate, which made this a nightmare. Luckily most of the logic had already been written in the non-platform-specific *master* branch I had prepared, so most of this sprint consisted of testing runtime logic.

11.3.2 Grove Framework

11.3.2.1 Overview

The Grove framework proved much easier to develop than the RN2483 library since the logic was much simpler (the complexity was compensated for by the design).

11.3.2.1.1 Hardware Abstraction

I managed to fit all the hardware specific functionality into a single namespace file – *GroveCommon*. This namespace contains functionality common to all the classes, which includes all the hardware-specific functionality. Consequently, porting the framework is incredibly easy since this is the only file that requires changes.

11.3.2.1.2 Naming convention

Aside from this, the only other thing to note is how I used simple class names instead of a factory class, which was a good decision in the end as calling upon the modules is now much simpler.

```
GroveCommon::mBit = &uBit;
GroveButton button(MICROBIT_PIN_P1);
int val = button.read();
```

Figure 31. Example of how easy it is to use the Grove Framework on the BBC Micro:Bit branch (first line is specific to Micro:Bit)

```
inc/:
total 64
-rw-r--r-- 1 alex alex 2634 Apr  2 15:03 GroveActuator.h
-rw-r--r-- 1 alex alex 2402 Apr  2 15:03 GroveButton.h
-rw-r--r-- 1 alex alex 1799 Apr  2 15:03 GroveBuzzer.h
-rw-r--r-- 1 alex alex 5204 Apr  2 15:04 GroveCommon.h
-rw-r--r-- 1 alex alex 2537 Apr  2 15:03 GroveCommunication.h
-rw-r--r-- 1 alex alex  693 Apr  2 15:03 GroveDrivers.h
-rw-r--r-- 1 alex alex 3007 Apr  2 15:03 GroveGPS.h
-rw-r--r-- 1 alex alex 2197 Apr  2 15:03 GroveLED.h
-rw-r--r-- 1 alex alex 1497 Apr  2 15:03 GroveLight.h
-rw-r--r-- 1 alex alex 2762 Apr  2 15:03 GrovePinout.h
-rw-r--r-- 1 alex alex 2034 Apr  2 15:03 GroveRotaryAngle.h
-rw-r--r-- 1 alex alex 1873 Apr  2 15:03 GroveSensor.h
-rw-r--r-- 1 alex alex 1497 Apr  2 15:03 GroveSound.h
-rw-r--r-- 1 alex alex 1914 Apr  2 15:03 GroveTemperature.h
-rw-r--r-- 1 alex alex 1606 Apr  2 15:03 GroveTouch.h

src/:
total 48
-rw-r--r-- 1 alex alex  590 Apr  2 15:03 GroveButton.cpp
-rw-r--r-- 1 alex alex  589 Apr  2 15:03 GroveBuzzer.cpp
-rw-r--r-- 1 alex alex 7328 Apr  2 15:04 GroveCommon.cpp
-rw-r--r-- 1 alex alex 1451 Apr  2 15:04 GroveGPS.cpp
-rw-r--r-- 1 alex alex  571 Apr  2 15:03 GroveLED.cpp
-rw-r--r-- 1 alex alex  497 Apr  2 15:03 GroveLight.cpp
-rw-r--r-- 1 alex alex  762 Apr  2 15:03 GrovePinout.cpp
-rw-r--r-- 1 alex alex  664 Apr  2 15:03 GroveRotaryAngle.cpp
-rw-r--r-- 1 alex alex  584 Apr  2 15:03 GroveSound.cpp
-rw-r--r-- 1 alex alex  955 Apr  2 15:03 GroveTemperature.cpp
-rw-r--r-- 1 alex alex  498 Apr  2 15:03 GroveTouch.cpp
```

Figure 32. Grove Framework class layout

11.3.2.1.3 Etc

The rest of the frameworks implementation follows **9.2**.

11.3.2.1.4 Examples

As noted in **11.3.3**, I ended up with a demo using just the Grove GPS module. Since it was so simple, I created a demo for each supported module in the framework (see *examples/* on the *platform/mbit* branch of the Grove Drivers repository).

11.3.2.2 *Sprint logs*

11.3.2.2.1 S8-R0, 26th March 2018

This was initially a lot more difficult than developing the RN2483 library since I'm less familiar with C++. Once I'd figured out the base classes though, the rest was a breeze.

In terms of portability, the final product has turned out to be much *more* portable than the RN2483 due to how classes force modularisation in C++. I made a few amendments to the design (see **9.2/Figure 21**) and think it's better than the post-prototype design (*although it did force me to cross lines on the diagram*).

11.3.3 Demo

11.3.3.1 *Overview*

This part of the project proved more difficult than the rn2483 library and grove framework (mostly a result of the Micro:Bit's Serial).

11.3.3.1.1 Design swap

The MicroBit's Serial caused serious issues in communicating with both the RN2483 module *and* the Grove GPS module, as they both required UART communication. I was able to communicate to the modules individually, however whenever trying to swap between them in the same runtime, communication stopped. I believe this was the result of a buffer issue, however at this stage of the project I didn't have time to investigate and resolve the issue, so instead I fell back to the original design of the demo in **Figure 22**. The shield I'm not using only has one digital/analog I/O port, but the Micro:Bit has an on-board temperature sensor that I was able to use. Using a button was slightly redundant however as the Micro:Bit has two onboard buttons, but its purpose was to demonstrate usage of the Grove Framework – which it achieves.

11.3.3.1.2 Compiling

The other *severe* problem I had was compiling. I initially tried to include the framework and library as dependencies or extra includes, however the Micro:Bit's CMake yotta build process made this much more difficult than it would be in standard projects and in the end, it become more of a headache than it was worth. Instead I just added the source and header files to the source directory of the yotta project and it worked fine.

11.3.3.1.3 Additional components

In addition to the demo code running on the Micro:Bit (which follows the design in **9.2**), I knocked up a simple nodejs script that makes a https GET request to the *Data Storage* application integration on The Things Network and then prints the retrieved data in the console(See **Figure 33**).

```
No data to retrieve
fetching data... student-diss.data.thethingsnetwork.org/api/v2/query/demo
No data to retrieve
fetching data... student-diss.data.thethingsnetwork.org/api/v2/query/demo
[2018-04-05T17:57:31.795377815Z] 26
```

Figure 33. Demo console log (before & after sending data)

11.3.3.2 Sprints

11.3.3.2.1 S6-R0, 2nd April 2018

I approached this by writing individual demos for the two components of the demo (LoRa & GPS). This worked fine, the GPS was particularly easy. I've added these individual demos as examples for their respective repositories.

Unfortunately, the Micro:Bit struggles when it has to hop between Serial lines (both the Grove GPS and RN2483 modules require UART communication), I was able to communicate with the RN2483 and GPS separately, however whenever I had to swap the serial from the RN2483's pin to the GPS's pins (or vice versa) more than once, the Serial communication would stop responding.

Instead I had to go back to the original demo design, where I sent temperature over LoRa whenever a button was pressed. While the Micro:Bit shield only had one Grove I/O port, it has an on-board temperature sensor that I was able to use.

11.3.3.3 Distance Testing

Despite not being able to use the GroveGPS module in combination with the RN2483 module, I was able to get a *rough* idea of the range that the RN2483 could achieve in my own environment.

I achieved this by sending data at intermittent locations and recording the time I sent the data. I'd then be able to view whether the data was received on the TTN and compare the tx/rx times with the times I'd recorded when sending the data. Unfortunately, the results were slightly disappointing.

Data was only sent successfully up to 0.2km in distance from my apartment. There are several possible reasons for this though, the main one being that I live in city centre of Bristol where there's a lot of interference. There's also the consideration that my apartment is at the top of its complex (7th floor) and that the Gateway I was using didn't have a necessarily large antenna. On top of this, the gateway was inside my apartment as I was testing, which mean that the LoRa packets would have had to travel through several walls.

One good note on the distance results was that they were still able to be sent/received reliably and uncorrupted in the distances I did achieve. Considering the number of physical surfaces, the packets would have travelled through and the amount of interference in the area, this is an impressive feat.

12 EVALUATION

12.1 PROBLEMS & FAILURES

The main hurdles/failures faced during project were:

1. The inability to setup the LoRaWAN network during Stage 1.
2. The failure in resolving the issues on the UWESense microcontroller board.
3. The inability for the BBC Micro:Bit to reliably redirect its Serial.

Despite these hurdles/failures, the only 2 and 3 caused a difference in the deliverables, and only 3 had a negative impact on the results of those deliverables.

Not being able to setup the LoRaWAN network early in the project (during Stage 1) was the first issue that occurred during the implementation. It was easily resolved, however did cause a delay in testing the tx/rx capabilities of LoRa and getting familiar with the process of sending/receiving data over a LoRaWAN network. Setting up a LoRaWAN network earlier on would have allowed for running more tests on the distance LoRa can achieve and provide earlier tests to compare with the tests from the final stage. While this had a negative impact on the research into the capabilities of LoRa, it didn't stop testing during the final stage (documented in **11.3.3.3**). Better prep work for the project would have mitigated this issue.

The UWESense was the main failure of the project. It might have been possible to resolve the issue with more time dedicated to working on the issue, however after 3 weeks of trying to resolve the issue (without much progress), the decision was made to change the target to complete the project on time. Whilst changing the target platform was evidently the right decision regarding project completion, the fact that even the cause of the issue remained a mystery is disappointing. Although, even spending 3 weeks on/off the same issue with little in the way of results is impractical, especially when there's the larger scope of the project to consider. This situation is a good example of when project management is necessary.

The BBC Micro:Bit's Serial was the final issue, this is the problem that caused a negative impact on the result of the deliverables[37]. The issue was likely an intricacy of the hardware/software (and could have been resolved with more time) however as this issue occurred at the end of the project (during the last sprint), another 'project manager' decision was made to simply revert back to the previous design (**Figure 22**) as the priority goal of the demo was to demonstrate the RN2483 library and Grove Driver framework. This meant that GPS data wasn't used in **11.3.3.3** and instead the results of testing were gained in a much more unreliable fashion, not only this but **requirement 4** was consequently not met.

[37] Arguably it was a by-product of issue 2.

With less time spent failing to resolve issue 2, this issue likely would have been overcome (this is a demonstration of where a more assertive project manager might have been better).

12.2 Improvements

The first group of improvements that could have been made are in the preparatory stage of the project. One of these improvements is in the research (**6**), specifically research into other potential target platforms for the demo (**6.2.1**) outside of the UWESense. This would have allowed for better preparation in the contingency that occurred regarding the UWESense. Not only this but it would have allowed for a more informed and logical decision in the MCU board used as a fall back, potentially one that used the same MCU (*STM32F401*) which would have kept the research covering it (**6.2.1.4**) relevant in the scope of the project. *In this regard, having a contingency plan for the project could have improved the overall result massively.*

Another element of the research that could have been improved was in the research of LoRa/LoRaWAN (**6.3**); no research was done into the ranges it was capable of, this improvement is related to the testing of LoRa/LoRaWAN and would have provided the project with more focus towards it and a base for any hypothesis. As it is, the testing of LoRa/LoRaWAN ended up being a *'P.S.'* of the project.

A preparatory element that wasn't carried out brilliantly was hardware and environment setup. Making sure that the required resources would be available (and tested!) before commencing the project would have gone a long way to avoiding the problems/failures covered in **12.1.2**. This would also have required that the project had a much more focused and clear goal from the beginning; as it was the project evolved slightly from its initial goal.

The methodology used in the project could also have been improved. While there's no doubt that TDD was the right decision for the project, the choice to keep review logs might have been unnecessary as the content of the review logs ended up as more of a mirror for the documentation in **11**. This certainly isn't a detractor (as it can be used to see how decisions were made, why and when), but it does add noise to that section which might not be necessary in all cases.

As well as the review logs, the testing could have been more thorough. Part of the problem in writing the test cases was that a lot of them were written based on vague Sprint descriptions and others were written after the sprint. In the latter case it didn't leave time for considering all the possible testing that could have been done and the former didn't provide a precise enough idea of the cases that would need to be passed to thoroughly test the relevant item. Despite this, using TDD in the actual development (making sure something as minor as a return value is correct before continuing) should have mitigated a lot of these issues (hopefully), but without clear documentation of this it means that

for the deliverables to be considered bug-free they'll need to go through a more thorough testing process – possibly with the eyes of a third-party.

The implementation was one of the strongest parts of this project, but the implementation of serial communication on the *platform/mbit* branch of the RN2483 library ended up being slightly messy, requiring the using to pass their MicroBit instance's Serial object to every function. This *does* avoid potential issues with global variable naming, however I think the solution used in the Grove Framework was much tidier. Simply creating a *extern* pointer in the header named *RN2483_mBit* that the user is required to set to point to their MicroBit instance that the rest of the program calls on would have worked as a solution this this. Of course, the currently implementation removes the possibility of the user forgetting to do this and have runtime issues with the library.

12.3 Achievements & Successes

The most successful part of the project was certainly in the design and how it evolved throughout the project. Making the decision to test the designs in **Stage 1** and trying to build them around being flexible was a very good – especially regarding the Grove Framework, which was going to need to be a flexible design from the outset but became more so. Not only was this but the flexibility of the Grove Framework design got tested (successfully) with the addition of the *GroveCommunication* and *GroveGPS* classes.

The number of requirements met for this project was also one of its feats. While not all were met, and some were only partially met, the majority of them were achieved. Some were even achieved beyond their original intention. An example of this is in the decision to use Git for version management (**requirements 13, 14**), which was especially helpful responsible for the idea of branching different versions would have come about.

The final implementation also ended up being a better result than expected. While not completely portable, modularising the non-portable elements of each source code and then branching the source-code to have specific implementations of these non-portable elements provides deliverables 1 and 2 with a lot more future-proofing and compatibility than was initially planned. The speed at which progress was made during the sprints was another good point on the implementation, the use of TTD methodology helped find and solve small issues while they were small, allowing to be made much more smoothly.

12.4 Future Work

The most obvious next step in future work will be creating more branches of the RN2483 Library and Grove Framework repositories to make them available on more platforms.

Before this can be done, it'll be beneficial to test the branch working on the Micro:Bit further and iteratively test and improve the *master* branch so that any problems or bugs that have been missed in it can be resolved.

Aside from this, the RN2483 library might require some more modularisation and error reporting, or at least a better way of reporting errors than *just* return codes, as porting it to the Micro:Bit didn't provide an opportunity to test how error-safe the code is.

Beyond this, adding support for as many Grove Modules as possible should be an end-goal for the Grove Framework. Given the design, it shouldn't be very difficult either, since any module implementation added to the *master* branch can just be pulled into the *platform/* branches without requiring any merge conflicts (unless it's in the *GroveCommon* file).

12.5 Closing Statement

The project was an overall success and delivered its 3 main deliverables (declared in **5.1**), not only this but the problems that occurred were overcome without compromising what those deliverables provide.

The possibility of both the RN2483 library and Grove Framework being available on multiple platforms that currently lack the support for these modules is exciting, not only this but it's a very contender as an open-source project since anybody who needs it can add support for their specific platform and contribute that to everybody else.

Hopefully they'll provide some level of familiarity to embedded developers across any supported platforms.

13 Bibliography

Alexandrescu, A., 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied.* 2nd ed. s.l.:Addison Wesley.

Amazon, 2017. *What is Cloud Computing?.* [Online]
Available at: https://aws.amazon.com/what-is-cloud-computing

Arduino, n.d. *Introduction.* [Online]
Available at: https://www.arduino.cc/en/Guide/Introduction
[Accessed 28 December 2017].

Atzori, L., Iera, A. & Morabito, G., 2010. The Internet of Things: A survey. *Computer Networks,* 54(15), pp. 2787-2805.

Barr, M. & Massa, A., 2006. *Programming Embedded Systems.* 2nd ed. s.l.:O'Reilly Media.

Battle, S. & Gaster, B., 2017. *LoRaWAN Bristol.* Bristol, s.n., p. 4.

CERP-IoT - Sundmaeker, H., Guillemin, P., Woelfflé, S. & Friess, P., 2010. *Vision and Challenges for Realising the Internet of Things.* Luxembourg: s.n.

Dieter Uckelmann, M. H. F. M., 2011. An Architectural Approach Towards the Future Internet of Things. In: *Architecting the Internet of Things.* Heidelberg: Springer, pp. 1-24.

Dongare, A. et al., 2017. *OpenChirp: A Low-Power Wide-Area Networking Architecture.* Kona, hl, pp. 569-574.

Ernst, R., Henkel, J. & Benner, T., 1993. Hardware-software cosynthesis for micrcontrollers. *IEEE Design & Test of Computers,* 10(4), pp. 64-75.

Espotel, 2015. *LoRaWAN Certification Test Report.* Jyväskylä: Espotel Oy.

Evans, D., April 2011. *The Internet of Things How the Next Evolution of the Internet Is Changing Everything,* s.l.: Cisco.

Ferran Adelantado, X. V. P. T.-P. B. M. J. M.-S. T. W., 2017. Understanding the Limits of LoRaWAN. *IEEE Communications Magazine,* January.

Gluhak, A., 2017. *Low Power Wide Area Networks: The new backbone for the Internet of Things.* [Online]
Available at: http://www.theiet.org/sectors/information-communications/topics/ubiquitous-computing/articles/lpwan.cfm
[Accessed 28 October 2017].

Google, 2017. *Internet of Things Articles.* s.l.:s.n.

Grenning, J. W., 2011. *Test-Driven Development for Embedded C.* Raleigh, North Carolina (Dallas, Texas): The Pragmatic Bookshelf.

Gridling, G. & Weiss, B., 2007. *Introduction to Microcontrollers.* 1.4 ed. Vienna: Vienna University of Technology Institute of Computer Engineering Embedded Computing Systems Group.

Gutwein, D., 2016. *Intel Announces First IoT Platform Designed for Retail.* [Online]
Available at: https://blogs.intel.com/iot/2016/01/19/intel-announces-first-iot-platform-designed-for-retail/
[Accessed 29 December 2017].

Haller, S., 2010. *The Things in the Internet of Things,* Zurich: SAP Research Center Zurich.

Hornbuckle, C. A., 2008. *Fractional-N Synthesized Chirp Generator.* Torrance, CA (US), Patent No. US 7,791,415 B2.

Inoue, A. et al., 1998. *A Programming Language for Processor Based Embedded Systems,* s.l.: s.n.

Intel Corporation, 2015. *Intel® Retail Sensor Platform Solution Brief.* s.l.:Intel Corporation.

Intel Corporation, 2015. *RFID and End-to-End Retail Analytics.* s.l.:Intel Corporation.

ISO/IEC, 2006. *Technical Report on C++ Performance,* s.l.: s.n.

ISO/IEC, 2006. *Technical Report on C++ Performance,* s.l.: ISO/IEC TR.

Iwata, J., 2012. *Making Markets: Smarter Planet.* s.l., IBM.

James Shore, S. W., 2007. *The Art of Agile Development.* First ed. s.l.:O'Reilly Media Inc..

Jayavardhana Gubbi, R. B. S. M. M. P., 2013. Internet of Things (IoT): A vison, architectural elements, and future directions. *Future Generation Computer Systems,* September, 29(7), pp. 1645-1660.

Knight, M., 2016. *Decoding the LoRa PHY.* s.l.:33c3: works for me.

Leverege, 2016. *LoRa & LoRaWAN Primer,* s.l.: s.n.

Leverege, 2016. *LPWAN White Paper,* s.l.: Leverage.

LoRa Alliance, 2015. *LoRaWAN™ What is it? A technical overview of LoRa® and LoRaWAN™,* s.l.: s.n.

martin, B., n.d. *TheThreeRulesOfTDD.* [Online]
Available at: http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
[Accessed 28 01 2018].

Meola, A., 2016. *These are the most powerful Internet of Things companies.* [Online]
Available at: http://uk.businessinsider.com/these-are-the-most-powerful-internet-of-things-companies-2016-7
[Accessed 29 December 2017].

Meulen, R. v. d., 2017. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. *Gartner Newsroom,* 7 February.

Microchip, 2016. *Low-Power Long Range LoRa® Technology Transceiver Module.* Revision C (April 2017) ed. s.l.:s.n.

Microchip, 2017. *DS40001784F.* s.l.:s.n.

Miorandi, D. S. S. F. D. P. I. C., 2012. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks,* September, 10(7), pp. 1497-1516.

Murphy, N., 2016. *How to Allocate Dynamic Memory Safely.* [Online]
Available at: after reading this article: https://barrgroup.com/Embedded-Systems/How-To/Malloc-Free-Dynamic-Memory-Allocation
[Accessed 20 03 2018].

N. Sornin (Semtech), M. L. (. T. E. (. T. K. (. O. (., 2015. *LoRa Specification.* V1.0 ed. s.l.:LoRa Alliance.

seeed studio, n.d. *Grove System Introduction.* [Online]
Available at: http://wiki.seeed.cc/Grove_System/
[Accessed 13 02 2018].

Shvets, A., n.d. *Design Patterns Explained Simply.* s.l.:Source Making.

Sign, A. K., 2008. Introduction to Microcontrollers. In: *Microcontroller and Embedded System.* 1st ed. s.l.:New Age International, pp. 1 - 12.

Sikken, B., 2018. *DecodingLoRa.* [Online]
Available at: https://revspace.nl/DecodingLora
[Accessed 02 03 2018].

ST, 2017. *Description of STM32F4 HAL and LL drivers.* Revision 5 ed. s.l.:ST.

Standard C++ Foundation, 2017. *Standard C++ FAQ.* [Online]
Available at: https://isocpp.org/wiki/faq/multiple-inheritance#mi-diamond
[Accessed 29 January 2018].

ST, n.d. *STM32 32-bit ARM Cortex MCUs.* [Online]
Available at: http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html
[Accessed 28 01 2018].

ST, n.d. *STM32F401.* [Online]
Available at:
http://www.st.com/en/microcontrollers/stm32f401.html?querycriteria=productId=LN1810
[Accessed 28 01 2018].

Stroustrup, B., 2017. *Bjarne Stroustrup's FAQ.* [Online]
Available at: http://www.stroustrup.com/bs_faq.html
[Accessed 15 02 2018].

Swedberg, C., 2015. Intel Unveils RFID System for Retailers. *RFID Journal,* 18 September.

Tara Salman, R. J., 2017. A Survey of Protocols and Standards for Internet of Things. *Advanced Computing & Communications,* 1(1).

Verizon, 2017. *State of the Market: Internet of Things 2017,* s.l.: Verizon.

Wayne C. Booth, G. G. C. J. M. W., 2008. *The Craft Of Research.* Third Edition ed. Chicago: The University of Chigaco Press.

Xia, F., Yang, L. T., Wang, L. & Vinel, A., 2012. Internet of Things. *International Journal of Communications Systems,* Issue 25, pp. 1101-1102.

14 Appendices

14.1 Defining 'Internet of Things'

It's easy to use somebody else's definition, however there's almost one definition for every paper written about IoT – each with its own variation on the concept. To avoid confusion, *another* variation will be defined here, however it will be the result of reviewing and concluding several definitions in the hope that a clear definition will come to fruition because of discussions like this.

To begin, it's important to breakdown the term and decide what about it requires defining. *"the name "Internet of Things" itself, which syntactically is composed of two terms."* (Atzori, et al., 2010), these two terms are of course 'Internet' and 'Things'. Whilst a definition of the term requires that these two work in conjunction, it offers those defining the term the option of focusing on a network-oriented vision, or an object-oriented vision.

As mentioned, one option is to take a 'network-oriented' approach when creating a definition. This approach focuses on the interconnection of devices and the infrastructure of the vision. For example, Atzori 's definition: *"a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols"* (Atzori, et al., 2010) tells us little about what 'things' are and what they'll communicate, instead it provides a very strong vision of how the infrastructure could work and how the objects will communicate.

On the opposite side of the table, is the focus on an 'object-oriented' vision, that focuses on the 'Things' half of the term. A typical 'object-oriented' approach is often much more futurist, e.g. *"a world where physical objects and beings, as well as virtual data and environments, all interact with each other in the same space and time"* (CERP-IoT - Sundmaeker, et al., 2010), this definition is much less specific in practicality than the previous, but provides a better idea of what the 'Things' are and what they might communicate.

Regardless of the approach taken in defining 'Internet of Things', it's important to remember that these definitions are not competing but instead adding pieces to a puzzle. *"All the different definitions of the term "Internet of Things" have in common that it is related to the integration of the physical world with the virtual world of the Internet."* (Haller, 2010). So, a good definition would be one that emphasises and provides clarity on this, while trying to keep a balance between 'Internet', 'Things', a vision, and practicality.

The very common (go-to) definition that is often seen today is something along the lines of a *"networked interconnection of everyday objects, which are often equipped with ubiquitous intelligence"* (Xia, et al., 2012), this is an acceptable definition, although it leaves a lot of gaps, it's also not too much of a far cry from the *"original concept proposed by Kevin Ashton at the Auto-ID*

center [1] of an informational network that allows the lookup of information about real-world objects by means of a unique ID" (Haller, 2010).

"A global network of interconnected, uniquely addressed objects, that communicate information and interact intelligently across the internet."

While this definition certainly doesn't solve all the problems presented and leaves a lot of unanswered questions[38], it provides a stable foundation for the rest of these writings and combines interrelated aspects of the reviewed definitions.

[38] I.e. Does 'Internet' refer to *the* internet or a new type of IoT 'internet'?

14.2 Previous Grove Designs

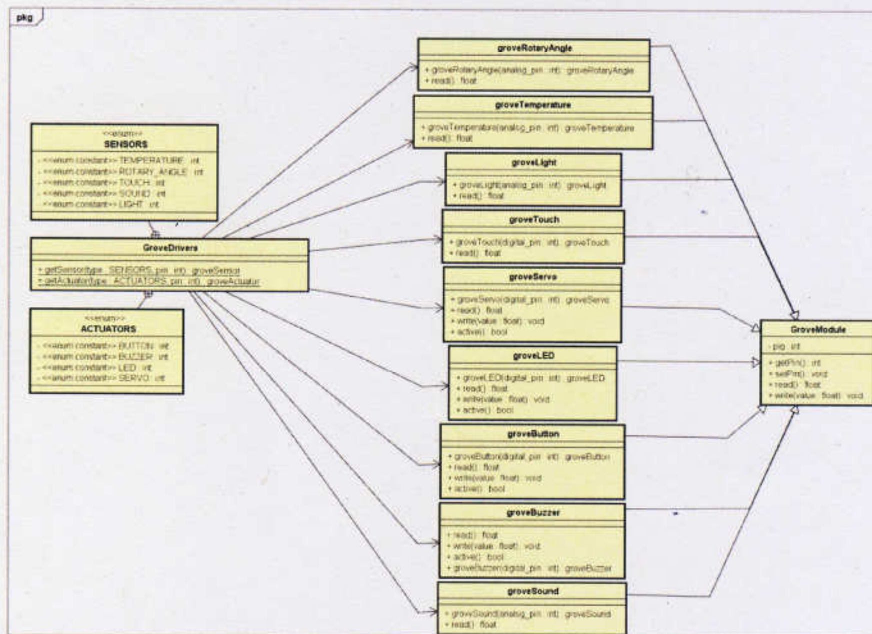


Figure 34. Initial design (before Stage 1)

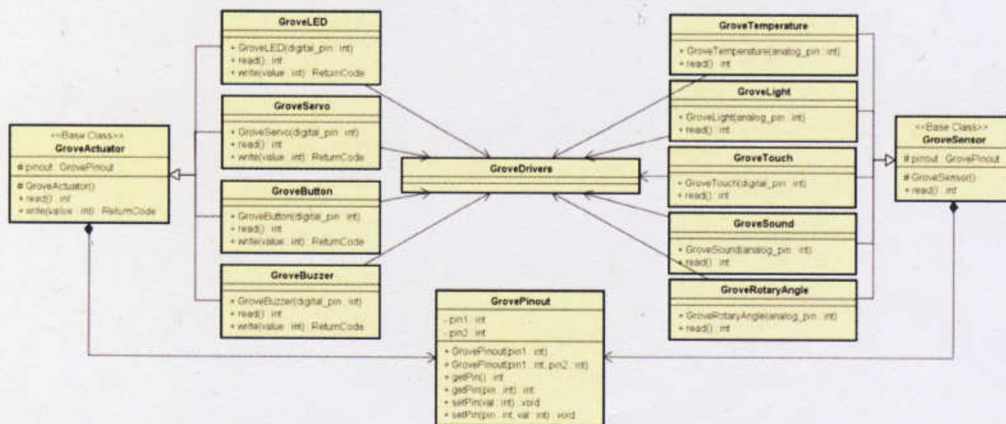


Figure 35. Second design (after Stage 1)