

Investigation of Connectivity, Energy Consumption and Real-time Properties in a LoRa-Network, using Vibration Sensors as case

Tobias Ulfsnes Rasmussen

February 2020



NTNU – Trondheim
Norwegian University of
Science and Technology



MASTER THESIS DESCRIPTION

Candidate:	Tobias Ulfsnes Rasmussen
Course:	TTK4900 Engineering Cybernetics
Thesis title (Norwegian)	Undersøkelse av konnektivitet, energi forbruk og sanntids-egenskaper i et LoRa-nettverk, med vibrasjonsensorer som bruksområde
Thesis title (English):	Investigation of connectivity, energy consumption and real-time properties in a LoRa-Network, using vibration sensors as case

Thesis description: We want to investigate properties to a sensor network using LoRa as communication infrastructure. The network should have nodes geographically distributed, but the measurements from the nodes should be related. As case is chosen a network of vibration sensors.

The thesis shall include further development of a flexible LoRa node used in previous work, to be used as vibration sensor nodes. A complete link from the nodes to a server collecting the data should be demonstrated, including a simple alignment of the data.

The study should include communication properties, energy consumption, real time properties and in general, the adequacy of the network as sensor network.

The tasks will be:

1. Perform a literary survey concerning use of LoRa in sensor networks
2. Specify and develop a vibration sensor LoRa node adequate for being a sensor network node.
3. Suggest and implement an infrastructure for connecting the nodes to a server, displaying the collected data.
4. Investigate the properties of network of nodes as developed under point 2.

Start date: August 20th, 2019
Due date: February 11th, 2020

Thesis performed at: Department of Engineering Cybernetics
Supervisor: Professor Geir Mathisen, Dept. of Eng. Cybernetics

Abstract

Distributed measurement systems have been used in the industry for years and with great success. However new technologies in IoT opens up for new solutions in the world of control and sensory systems. LoRaWAN is one of many protocols made specifically for low-power IoT devices.

In this thesis an embedded vibration-sensor has been designed, tested and implemented, using LoRa as communication. A set of vibration-sensors has been deployed in order to simulate a DMS (distributed measurement system) using LoRaWAN as a communications protocol. This thesis investigates synchronization, power-consumption and connectivity properties of LoRaWAN, with a DMS as a use-case. The deployed system managed to successfully gather vibration data and store it on a remote server.

A full solution covering all aspects were never deployed. However, a DMS with LoRaWAN as communications were deployed and data was collected. Synchronization was done using LoRa and that proved to be a feasible solution for communications in DMS.

Sammendrag

Distribuerte målesystem har blitt brukt i industrien i flere år og med stor suksess. Nye teknologier innenfor IoT åpner opp for nye løsninger i verdenen av kontroll og målesystemer. LoRaWAN er en av mange protokoller laget spesielt for lav-energi IoT enheter.

En trådløs vibrasjon-måler som bruker LoRa for kommunikasjon ble designet og implementert. Ett sett med vibrasjon-målere ble utplassert med det formål å simulere ett distribuert målesystem med LoRaWAN som kommunikasjon. Denne oppgaven undersøker LoRaWAN's egenskaper innenfor synkronisering, energi-forbruk og konektivitet. Løsningen var i stand til å samle inn vibrasjonsdata fra forskjellige sensorer og lagre dataen på en ekstern server.

En full løsning ble aldri implementert og testet. Ett distribuert målesystem med LoRaWAN som kommunikasjon ble deployert og data innsamlet. LoRa ble brukt til å synkronisere sensorer og det ble bevist å være en mulig løsning for kommunikasjon i et distribuert målesystem.

Preface

This thesis is submitted in fulfillment of Master of science at the Department of Engineering Cybernetics, Norwegian University of Science and Technology. The work done in this master thesis continues the work done in the TTK4450-project.

The embedded system developed is inspired by the system developed in the specialization project. All of the work presented in this paper is done by me between the start and end date of the thesis. The thesis will provide trace-ability from the specifications and throughout the paper. This should make it possible for anyone to re-create the same experiments that were conducted by me.

Working with this thesis has been very challenging and a learning experience. I have gotten a unique insight into the world of embedded systems. Both in terms of hardware design and software development.

Acknowledgements

I want to thank my supervisor Geir Mathisen for guiding me throughout the entire thesis. And for engaging in interesting discussions along the way. Also I would thank the Employees at the Electronics workshop at the Department of Engineering Cybernetics for letting me use their equipment during development and implementation. Lastly I would thank Bente Eva and Torbjørn Ulfsnes for supporting me throughout my years at NTNU.

Approved: 

Prof. Geir Mathisen

Thesis supervisor

Approved: 

Tobias Ulfsnes Rasmussen

Author

Table of Contents

Thesis description	i
Abstract	ii
Sammendrag	iii
Preface	iv
Table of Contents	viii
List of Tables	ix
List of Figures	xii
Abbreviations	xiii
1 Introduction	1
1.1 Background	1
1.2 Motivation	2
1.3 Limitations	2
1.4 Thesis structure	2
2 Literature review	4
2.1 A lightweight synchronization algorithm	5
2.2 Connectivity limitations in LoRaWAN	7
2.2.1 Duty-cycle limitation	7
2.3 Power consumption in LoRa networks	7
2.3.1 Theoretical life-time expectancy of a LoRa device	9
3 Theory	10
3.1 Overview	10
3.2 LoRaWAN protocol	10

3.2.1	LoRa Basics	10
3.2.2	LoRa classes	11
3.2.3	MAC message format	12
3.2.4	Network join procedures	13
3.2.5	LoRa uplink sequence and timing	13
3.2.6	LoRa airtime	16
3.3	MQTT	18
3.4	NTP	18
4	Specification and Design	20
4.1	Overview	20
4.2	Functional requirements	22
4.3	Specification	23
4.3.1	End-device	23
4.3.2	Gateway	24
4.3.3	Server/Handler	25
4.4	Acceptance criteria	25
4.5	Design	26
4.5.1	PCB design	26
4.5.2	Back end	27
4.5.3	System design	28
5	Implementation	30
5.1	Hardware End-device	30
5.1.1	Components	31
5.1.2	Power circuit	33
5.1.3	Accelerometer circuit	34
5.1.4	RF transmitter circuit	35
5.1.5	Headers and peripherals	35
5.2	PCB result	36
5.3	Software End-device	37
5.3.1	RTC	37
5.3.2	Timer	38
5.3.3	Watchdog timer	39
5.3.4	Drivers	39
5.3.5	Accelerometer and temperature calibration	42

5.3.6	Collecting vibration data	42
5.3.7	RTC alarm	44
5.3.8	Main program	44
5.3.9	OTA Synchronization	50
5.4	Private gateway	52
5.4.1	LoRa network configuration	52
5.4.2	Gateway SW	52
5.5	RT server	54
5.5.1	Hardware	54
5.5.2	Software	55
5.6	TTN back-end	57
5.6.1	TTN console and SDK	57
5.6.2	MQTT-client	58
6	Testing and results	60
6.1	Test procedure	60
6.1.1	Hardware test	60
6.1.2	OTA synchronization	61
6.1.3	Battery lifetime expectancy	62
6.1.4	Vibration data-collection	63
6.1.5	End-device clock drift	64
6.1.6	End-device remote configuration	64
6.2	Results	64
6.2.1	OTA synchronization	65
6.2.2	Battery lifetime expectancy	69
6.2.3	Vibration data	73
6.2.4	Calibration results	78
6.2.5	Time server synchronization	78
6.2.6	End-device clock drift	79
6.2.7	End-device remote configuration	79
7	Discussion	81
7.1	System results	81
7.2	Synchronization and clock accuracy	83
7.2.1	End-device clock drift	83
7.2.2	Synchronization accuracy	84

7.3	Battery lifetime expectancy	84
7.4	Vibration data analysis	86
7.5	Areas of improvement	87
8	Conclusion	90
9	Further Work	91
	Bibliography	92
	Appendix A: The Things Network	95
	A1: Application server GUI	95
	A2: Device control panel	96
	Appendix B: HW End-device	97
	B1: Schematic	97
	B2: Part list	98
	B2: PCB trace	99
	Appendix C: Private gateway	100
	C1: Gateway LoRa configuration	100
	C1: Gateway join server	101

List of Tables

2.1	Timing requirements in typical DMT's	5
3.1	LoRa message types	12
3.2	Variable description for fig. 3.6	15
3.3	LoRa Data-rates	18
4.1	Functional specification	22
4.2	Specification HW	23
4.3	Specification SW	24
4.4	Specification Gateway	24
4.5	Specification server	25
4.6	Acceptance criteria.	26
5.1	Private gateway metadata	53
6.1	Alive message logged by the MQTT-client.	64
6.2	Vibration-data logged by the MQTT-client.	65
7.1	Variable description for eq. (7.4)	83

List of Figures

2.1	A lightweight synchronization algorithm	6
2.2	Congestion in LoRaWAN	8
2.3	Life-time expectancy in LoRaWAN	9
3.1	LoRa Class A receive-windows.	11
3.2	Uplink packet format	12
3.3	LoRa PHYPayload structure	13
3.4	LoRa MACPayload structure	13
3.5	Sequence diagram: LoRa uplink w/callback.	14
3.6	Timing diagram: LoRa uplink w/callback.	14
3.7	MQTT collaboration diagram	19
4.1	System collaboration	21
4.2	Block diagram PCB	27
4.3	System design with RT-server	29
4.4	System design with RT-server	29
5.1	Schematic: Power circuit	34
5.2	Schematic: Accelerometer circuit	34
5.3	Schematic: RF transmitter circuit	35
5.4	Assembled End-device	36
5.5	Flowchart: Accelerometer calibration	43
5.6	Flowchart: MPU6050 ISR	44
5.7	Flowchart: RTC ISR	45
5.8	State diagram: End device	46
5.9	Flowchart: Idle state	47
5.10	Message frame: Alive message	47
5.11	Flowchart: Alive transmit state	48
5.12	Message frame: Event message	49

5.13	Message frame: Append data message	49
5.14	Flowchart: Data transmit state	50
5.15	Flowchart: End-device OTA synchronization scheme	51
5.16	Message frame: Synchronization downlink	51
5.17	Snapshot: Gateway uplink forward procedure.	53
5.18	Snapshot: Gateway downlink forward procedure.	53
5.19	Snapshot: Gateway RTT response procedure.	54
5.20	RT-server collaboration diagram.	54
5.21	Flowchart: Time server synchronization.	56
5.22	Flowchart: Synchronization callback.	57
5.23	.csv Alive data format	58
5.24	.csv Event data format	58
6.1	OTA-synchronization test-setup.	61
6.2	Flowchart: End-device print-ISR	62
6.3	Illustration: Device placement	63
6.4	Illustration: Device axis description	63
6.5	Time series: Synchronization-error DR=5	65
6.6	[Time series: Synchronization-error stable region DR=5	66
6.7	Time series: Synchronization-error improved DR=5	66
6.8	Time series: Synchronization-error improved stable region DR=5	67
6.9	Time series: Synchronization-error DR=0	67
6.10	[Time series: Synchronization-error stable region DR=0	68
6.11	Time series: Synchronization-error improved DR=0	68
6.12	Time series: Synchronization-error improved stable region DR=5	69
6.13	Power consumption during alive message.	70
6.14	Power consumption while sending vibration data.	70
6.15	Power consumption while in sleep mode at 1.25 Hz sample rate.	71
6.16	Power consumption while in sleep mode at 1.25 Hz sample rate.	71
6.17	Power consumption during wake-up.	72
6.18	Power consumption when going to sleep.	72
6.19	Power consumption while transmitting with $DR = 0$ and $DR = 5$	73
6.20	Time series: Vibration data x-axis	74
6.21	Time series: Vibration data y-axis	75
6.22	Time series: Vibration data z-axis	75

6.23	Time series: Vibration data device 3	76
6.24	Time series: Light level device 1	77
6.25	Snapshot: Calibration result	78
6.26	Snapshot: Time server synchronization	78
6.27	Snapshot: Clock drift test start.	79
6.28	Snapshot: Clock drift test end.	79
6.29	Snapshot: End-device remote configuration.	79
6.30	Snapshot: Back-end remote configuration.	80

Abbreviations

IoT	=	Internet of Things.
MAC	=	Medium Access Control.
LoRa	=	Modulation-standard for IoT.
LoRaWAN	=	A Wide Area Network MAC-layer protocol.
DMS	=	Distributed measurement system.
UDP	=	User Datagram Protocol.
RTT	=	Round trip time.
API	=	Application programming interface.
MQTT	=	A publish-subscribe architecture.
GUI	=	Graphical User Interface.
Uplink	=	An transmission from an end-device to a server/back-end.
Downlink	=	A transmission from server or back-end to a end-device.
DR	=	Data-rate.
CR	=	Cyclic coding rate.
CRC	=	Cyclic redundancy check.
SF	=	Spreading factor.
BW	=	Band-width.
GW	=	Gateway.
ACKNACK	=	Acknowledged/Not acknowledged.
ISR	=	Interrupt service routine.
WDT	=	Watchdog timer.
RTC	=	Real time clock.
ADC	=	Analog to digital converter.
AppEUI	=	Application identifier.
ABP	=	Activation by Personalization.
OTAA	=	Over the air activation.
MQTT	=	MQ telemetry transport.
OTA	=	Over the air.

PCB = Printable Circuit Board.
End-Device = A device which utilizes LoRa to communicate externally.
GUI = Graphic user interface.
NTP = Network time protocol.
SDK = Software development kit.

Chapter 1

Introduction

1.1 Background

Internet of things can be defined as the interconnection between embedded devices and the internet. The area of application is still not fully envisioned as every year new applications and areas of use are uncovered. A casual way of describing IoT is like a distributed set of embedded systems sending usable data like for instance temperature, to a backend. One such system is a wireless sensor network. Such a network collects valuable data from the surrounding environment in order to observe the said environment. These have been used to monitor agriculture, weather, air pollution, Forrest-fires, etc. Such sensors must have a long battery life since they might be placed on remote locations over long periods of time.

One area these can be used is for measuring seismic activity. These are vibration sensors that can be used to detect activity in exposed areas in order to warn in case of earthquakes or tsunamis. However, vibration sensors can be used in other applications. For instance, an important aspect of building and maintaining bridges is the frequency with which it oscillates. During the design, it is important to uncover that the most common frequencies induced are not in the area of the natural-frequency for the construction. Furthermore, the measured frequency can also be used to detect structural integrity. These can be measured by deploying measurement-devices that record vibrations over time, and are collected after a certain amount of time. A better solution could be to deploy a device that continuously provide data, and which is connected wireless. This would enable engineers to continuously

monitor the structural integrity of the construction.

Can a protocol designed for IoT was used to implement such a system? LoRaWAN is one such protocol and is designed especially for being low-power and is seemingly a candidate for such an implementation. Furthermore, such a system may have events that are time-related, which in turn calls for a way to align data according to the time they were collected.

1.2 Motivation

To work on something so new and inclusive as IoT is very interesting. New solutions and areas of applications pop up all the time, so to spear this development is very motivating indeed. Furthermore, to engage in the realization of an idea in embedded is at the very least intriguing. Investigating new application-areas and properties for LoRaWAN gives a sense of ownership.

1.3 Limitations

The sensors were never deployed out in the field, ideally, the vibration sensors would be deployed on a structure, for instance, a railway-bridge. Furthermore, the system would have been deployed using a private-network. The fact that only one gateway was available invokes limitations in network-coverage, therefore it was decided to use an external public network upon deployment.

1.4 Thesis structure

Chapter 2 Literature review contains a survey conducted on use of LoRaWAN in DMS, synchronization in LoRa networks and life time expectancy of LoRa-devices.

Chapter 3 Theory adds the theory needed in order to understand this thesis. It covers theory about LoRaWAN and MQTT.

Chapter 4 Specification and design present the functional and technical requirements in addition to the acceptance criteria for both software and hardware. The chapter also

contains a design for the distributed sensor network.

Chapter 5 Implementation contains details about the implementation of hardware software and the complete system. **Chapter 6 Testing and results** describes the testing procedure and corresponding testing-results.

Chapter 7 Discussion gives a further discussion of the results in addition to suggested improvements.

Chapter 8 Conclusion concludes the thesis and its findings.

Chapter 9 Further work contains a list of suggested further work.

Appendix A contains snapshots of TTN's application server-GUI and device control-panel.

Appendix B provides an HW-schematic and part-list for the embedded-system. **Appendix**

C contains snapshots of the private-gateway configuration and setup.

Chapter 2

Literature review

The scope of this paper is to investigate LoRaWAN as a communications infrastructure in a distributed measurement system(DMS). This section covers a literature review done on this subject. IoT is often defined as the collection of large amounts of data from a large set of sensors out there in the real world. This is virtually the same as a traditional DMS only deployed on a much larger scale, thus a question raised in the last years: Are IoT protocols suitable for traditional DMS? Furthermore is LoRaWAN as a protocol developed specifically for IoT suitable for traditional DMS's?[1]

"Since a DMS is involved with time-series collection, time-related FoM(Figures of merit) as refresh time, latency, and required time synchronization accuracy are particularly meaningful." - M. Rizzi, P. Ferrari, A. Flammini, and E. Sisinni.[1]

In other words: In distributed systems, it is of the utmost importance to have synchronization across end-devices/nodes. LoRaWAN provides no such solution by itself, this means that it is up to the developer to implement such a mechanism. In addition to solving problems with concurrency and causality, synchronization may be used to improve the reliability of channels implementing scheduled uplinks (Timed Division Multiple Access)[2].

The synchronization requirement for typical DMT's varies widely with the domain in which it is applied, the table in table 2.1 is taken from [1] and is a summary of the work done in an article by M. Kuzlu, M. Pipattanasomporn, and S. Rahman[3].

Application	Update time(s)	Latency(s)	Sync err.(s)	Data Size(B)	#Nodes
Industrial Automation(Factory)	<0.1	0.01	<10 ⁻⁴	<100	10 ²
Industrial Automation(Process)	<60s	<1	<0.01	<100	10 ³
Smart Building(e.g HVAC)	<600	<60	<1	<1k	10 ²
Home Automation(e.g lighting)	Event based	<60	<10 ⁴	<100	10 ²
Smart metering (electricity)	<600	<10	<1	100	10 ⁶
Smart metering (Gas)	4day	<3600	<60	100	10 ⁶
Smart Grid (PMU)	<0.01	<0.01	<10 ⁻⁶	<200	10 ²
Smart Grid (EV fleet)	4/day	<15	<60	<300	10 ²

Table 2.1: The typical timing requirements in traditional DMT's.[1]

2.1 A lightweight synchronization algorithm

A synchronization algorithm is a viable way to synchronize distributed systems, which also compensates for clock skew[2]. It utilizes hardware functionalities to compensate for clock skew and drift over time. This enables a device to use lower grade parts and still have accurate time stamping. L. Tessaro, C. Raffaldi, M. Rossi, and D. Brunelli[2] proposes the

simple algorithm in fig. 2.1. A summary of steps in fig. 2.1 can be listed as follows:

- A constant error margin is chosen.
- When the first synchronization message is received, the RTC is updated.
- At the next synchronization message, the received timestamp will be compared with the current one $e_s = t_{sync} - t_{new}$.
- If the error e_s is within the range of the error margin set, no operation is performed.
- If the error e_s is outside this range, the error is not acceptable and a correction will be performed.

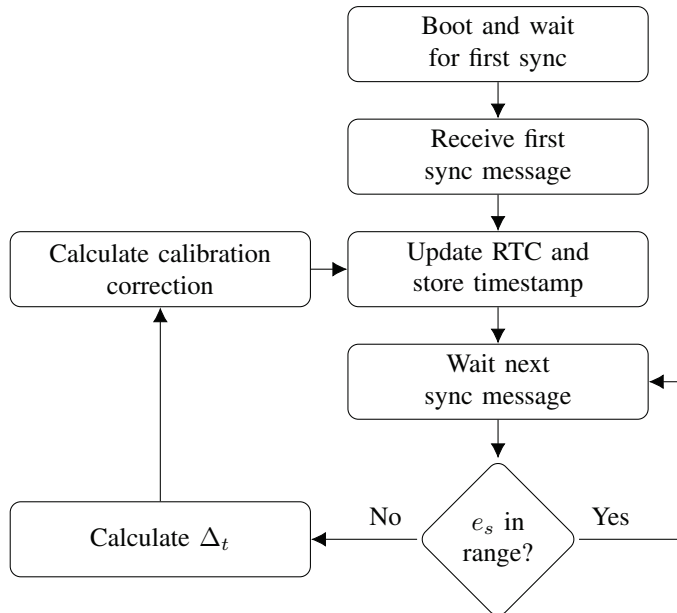


Figure 2.1: Synchronization algorithm for LoRa network[2].

With this synchronization scheme L. Tessaro, C. Raffaldi, M. Rossi, and D. Brunelli managed an average synchronization error(e_s) of 4.54 ± 1.28 ms.

2.2 Connectivity limitations in LoRaWAN

An important aspect in terms of RT use is the reliability of a protocol. More particularly the end-devices capability under various circumstances to reach a drain in the network. The main reasons for package loss between a GW and end-device are mainly that the drain is not reachable from the end-device. This is mostly caused by two factors: duty-cycle limitations and end-device RF power. The last of which is mostly restricted by the PHY-layer and MAC-layer. Duty-cycle, on the other hand, is a the key-limitation brought upon the protocol by the narrow-band channels(125 kHz[4]).

2.2.1 Duty-cycle limitation

The duty-cycle depends on the regional parameters set[4], and may vary from place to place. For the EU863-870 ISM band it is set to be $< 1\%$.

Given the time on-air T_a and duty cycle d , the minimum off period $T_s = T_a(1/d - 1)$ [5], an end-device in the EU can transmit for a maximum of 36 s/h. By exceeding the duty-cycle the limitation becomes clear, the channels will get congested. Take fig. 2.2 taken from F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Wat-tyne, as an example. The given 250 end-devices sending 2500+ packages an hour, of these packages only about 15% are received. That gives a package loss of about 75% due to end-devices exceeding the duty-cycle. Given the case that several devices in a network must air at virtually the same time, package loss is almost impossible to avoid.

2.3 Power consumption in LoRa networks

The promise and motivation behind LoRa-WAN are to have a Low-power RF-protocol which can work over long distances. A central aspect of an end-device is then the expected battery life of such a device.

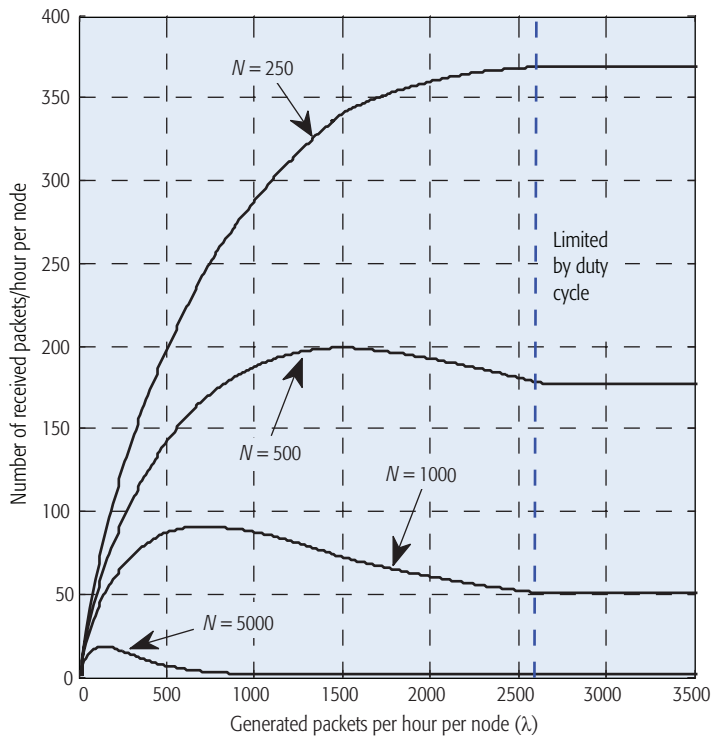


Figure 2.2: Number of 10 B payload packets received per hour and node for {250, 500, 1000, 5000} end devices[5].

2.3.1 Theoretical life-time expectancy of a LoRa device

Consider the following scenario: An end-device uses a 2000 mA/h battery and transmit for a total duration of 30 s/24 h using the LoRa modulation scheme. Furthermore each uplink-message containing a 10 B payload and is sent with $CR = 1$, $BW = 125$ kHz and $SF = 7 - 12$. The following numerical results has been provided:

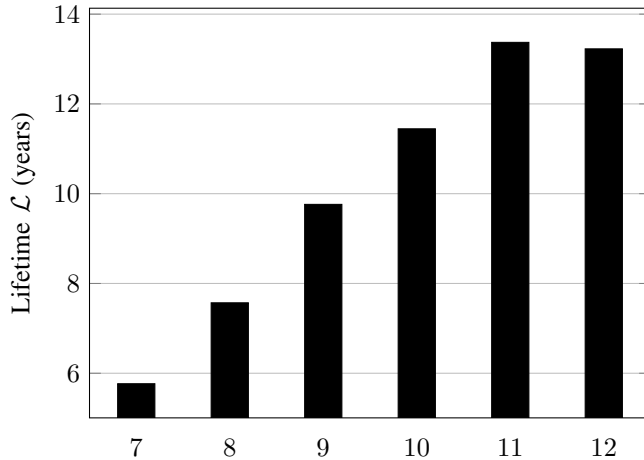


Figure 2.3: LoRaWAN life time expectancy for different spreading factors $SF \in \{7, 8, 9, 10, 11, 12\}$. Under the constraint of 30 s air-time per day and battery capacity of 2 A h[6].

These results are purely numerical and have not been field-tested. However can be used as an indication for how a device should perform, given the same configuration.

Chapter 3

Theory

3.1 Overview

This chapter intends to lay down basic theory relevant to the work done in the thesis. It provides a simple explanation of the LoRaWAN network and some theories about the LoRa-modulation. Most of the work in this chapter is a summary of theory presented in [7], [7] and [8]. section 3.2.5 is deduced by the writer of this paper after being inspired by the work done in [2]. Basic knowledge about LoRaWAN is required to understand the rest of the thesis.

3.2 LoRaWAN protocol

LoRaWAN is a network protocol optimized for battery-powered devices. This section is an amalgamation of information and theory taken from[7]. This provides a better understanding of what LoRaWAN is and how it operates.

3.2.1 LoRa Basics

LoRaWAN networks consist of three main entities: **End-devices**, **Gateways** and **Network-server**. An end-device is a device/sensor gathering information. This device will often be

battery-driven and can be active without maintenance over long periods. These devices will send the gathered information in an uplink(messages) to a gateway. The gateways relay information from the end-device to the network server. The gateway is often associated with base-stations or network-sinks. A network-server is where the information ends up, either to be processed or to be distributed(for instance through MQTT). The specification[7] treats the network-server, application-server(for instance MQTT broker) and join-server as co-located(one entity). An illustration of a simple network is provided in fig. 4.1.

3.2.2 LoRa classes

In a LoRa-network end-devices are classified into 3 different classes: Class A, B, and C. These give the end-device slightly different capabilities. Class A can only receive messages(downlinks) right after sending an uplink during defined receive-windows. There are two receive windows, if these are missed, no downlink is received. Class B devices can schedule downlink-windows i.e set a time of day where it will listen for downlinks. Class C has limitless receive windows. All classes offer bi-directional communication.

Receive windows

Following each uplink, end-devices must open two receive-windows. These windows have a defined delay between each other, this delay may differ from network to network, however for public networks they are defined in the Regional parameter specification[4]. fig. 3.1 is taken from an earlier project[9] and illustrates how the receive-windows work.

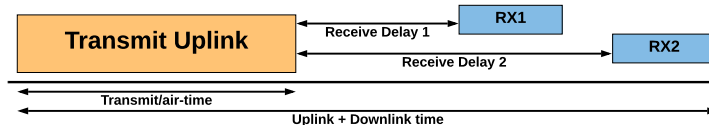


Figure 3.1: Illustration of the LoRa class A receive windows following an uplink transmission.

3.2.3 MAC message format

Each LoRa uplink and downlink has the format shown in fig. 3.2, with the exception for the payload CRC, this is only available on uplink-messages.

The payload(PHYPayload) of each uplink has a sub-structure comprising of: a MAC-header(MHDR), MAC payload(MACPayload) and a 4-octet message integrity code(MIC). An illustration of the structure can be seen in fig. 3.3. The MAC-Header contains information about what type of message this is. There are 7 different message types:

- Join-request
- Join-accept
- Unconfirmed data up
- Unconfirmed data down
- Confirmed data up
- Confirmed data down
- Rejoin-request

Table 3.1: Table listing the different types of LoRa messages.

Each MAC-payload(see fig. 3.4) can be seen as frames and comprises of: a header-field(FHDR), a port number(FPort) and lastly the payload(FRMPayload). This contains the actual data sent from the end-device, messages with different kinds of content can be sent on different ports. This way an application or server can easily differentiate between what kind of content each message contains. This is crucial since all information is sent as bytes.

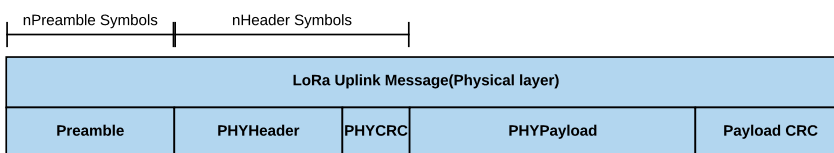


Figure 3.2: LoRa-uplink packet format.

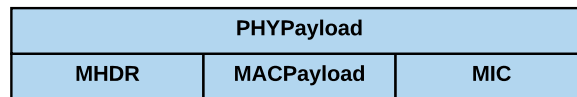


Figure 3.3: Physical payload structure.

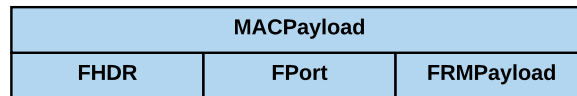


Figure 3.4: MAC payload structure.

3.2.4 Network join procedures

To join a network, each end-device has to be activated. This is done by sending a join-request which will be routed to a specific join-server. The join has a unique identifier called JoinEUI(applicationEUI), this is a IEEE EUI64 address. The join server holds the id(DevEUI) of every end-device attached to an application. The DevEUI is also a EUI64 address, unique for each device. Furthermore each network has a AES-128 root-key, specific for each device. These are often called AppKey's or NwkKey's. These are used to encrypt all messages, such that any uplink can only be read by the corresponding application or back-end. This ensure privacy both between end-devices, applications and networks. So to join a network, every end-device must have a DevEUI, AppKey and AppEUI. There are 2 different ways to join a network: **OTAA**(over-the-air activation) or **ABP**(activation by personalization). OTAA requires a unique DevEUI, AppKey and AppEUI for each device, and derive specific session-keys from these. Thus the session-keys can be dynamically updated. However ABP comes with pre-configured identifiers and session-keys from the factory, this means that they never need to join a network, however it also means that they have the same session-key throughout their lifetime. Thus for higher-security applications OTAA should be the preferred join procedure.

3.2.5 LoRa uplink sequence and timing

During a normal generic uplink with a corresponding downlink, a class A LoRa device will operate according to the sequence seen in fig. 3.5. Based on this diagram the different

timings/delays during such a sequence can be identified. This has been done in fig. 3.6, further explanation will follow beneath the figure.

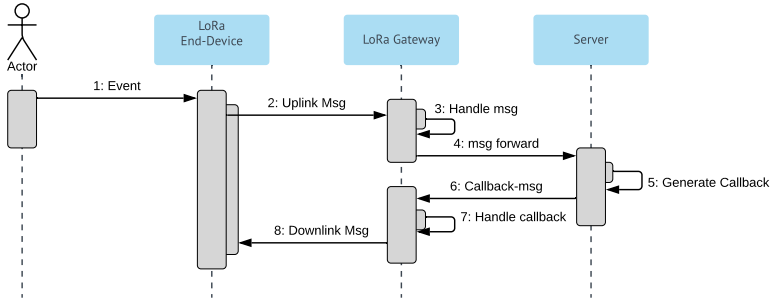


Figure 3.5: Sequence diagram showing the processes from a uplink to a corresponding downlink.

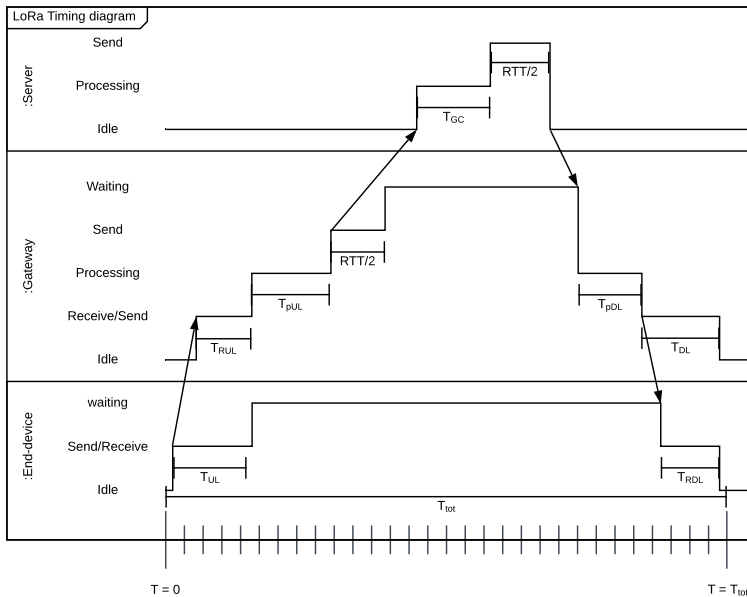


Figure 3.6: Timing diagram of a LoRa uplink and downlink.

To start the different delays/times are identified in table 3.2, they are presented in chronological order as presented in fig. 3.6.

T_{UL}	Time to air uplink
T_{RUL}	Time to receive uplink
T_{pUL}	Time to process uplink
$\frac{RTT}{2}$	Time to send a message from GW to server
T_{GC}	Time to generate a callback
T_{pDL}	Time to process downlink
T_{DL}	Time to air downlink
T_{RDL}	Time to receive downlink
T_{tot}	Total time from uplink sent to downlink received

Table 3.2: Table describing the different identifiers in fig. 3.6.

All of the variables in fig. 3.6 can be measured during operation by the use of for instance timers. This is except for the two air-times: T_{UL} and T_{DL} . Semtech has provided a way to calculate the airtime, this is further investigated in section 3.2.6. Continuing on the assumption that all variables except T_{DL} are known. T_{tot} can be written as:

$$T_{tot} = T_{pUL} + \frac{RTT}{2} + T_{GC} + \frac{RTT}{2} + T_{pDL} + T_{DL} \quad (3.1)$$

And assuming that all except T_{DL} is known:

$$T_{DL} = T_{tot} - \left(T_{pUL} + \frac{RTT}{2} + T_{GC} + \frac{RTT}{2} + T_{pDL} \right) \quad (3.2)$$

The total time from the uplink was engaged until the server sends the callback is:

$$T_{tot*} = T_{UL} + T_{pUL} + \frac{RTT}{2} + T_{GC} \quad (3.3)$$

The delay from T_{GC} to the message is received at the end-device is denoted as T_{skew} . This is the time passed from the timestamp was taken until it is received. This can be written as:

$$T_{skew} = \frac{RTT}{2} + T_{pDL} + T_{DL} \quad (3.4)$$

By using eq. (3.3), eq. (3.2) can be re-written as:

$$T_{DL} = T_{tot} - T_{tot*} - T_{pDL} - T_{DL} \quad (3.5)$$

And lastly putting eq. (3.5) into eq. (3.4):

$$T_{skew} = T_{tot} - T_{tot*} \quad (3.6)$$

With T_{skew} known, a potential timestamp during synchronization can be corrected when sent from the server. With the above scheme, an end-device could be synchronized with a server over the air.

3.2.6 LoRa airtime

This section focuses on the theoretical calculation of packet air-time. This is most commonly used to either calculate the duty-cycle for end-devices or estimate power consumption. The following theory is an amalgamation of standards found in the regional-parameters[4] for the EU and theory found in the SX1276-datasheet(section 4.1.1)[8].

The packet format is illustrated in fig. 3.2. To calculate the airtime, the number of symbols used in each these fields must be known. The preamble is a fixed at 8 Bytes, this is set by the regional parameters[4]. The header is optional, however in default mode in comprises of: Payload length in bytes, the code rate(CR) used and a 16-bit CRC for the payload. The payload length is dependant on the regional parameters and the data-rate(DR). selected[4].

Airtime calculation

The following is a summary of the airtime calculation given by Semtech[8].

Given the CR, Bandwidth(BW) and spreading factor(SF), the symbol duration(T_{sym}) can be defined as:

$$T_{sym} = \frac{2^{SF}}{BW} \quad (3.7)$$

Furthermore the time to air the preamble can be written as:

$$T_{preamble} = (n_{preamble} + 4.25)T_{sym} \quad (3.8)$$

Where $n_{preamble}$ is the number of preamble symbols(see fig. 3.2). The total symbol length of the packet payload and header is then given by:

$$n_{payload} = 8 + \max \left(\text{ceil} \left[\frac{8PL - 4SF + 28 + 16CRC - 20H}{4(SF - 2DE)} \right] (CR + 4), 0 \right) \quad (3.9)$$

- PL is the number of payload bytes.
- SF is the spreading factor (7-12).
- $H = 0$ if the header is enabled.
- $DE = 1$ when LowDataRateOptimize is enabled.
- CR is 1-4 dependant on which is selected.
- CRC is 1 if payload CRC is enabled.

Following the EU863-870 MHz ISM Band[4], the following can be defined:

$$\begin{aligned} H &= 0 \\ CR &= 1 \\ CRC &= 1 \end{aligned} \quad (3.10)$$

Using 3.10 and assuming that no uplink is sent without a payload, eq. (3.9) can be simplified:

$$n_{payload} = 8 + 5 \left(\text{ceil} \left[\frac{8PL - 4SF + 24}{4(SF - 2DE)} \right] \right) \quad (3.11)$$

Which can be used for all default LoRa devices following the EU863-870 MHz frequency-plan, where the standard channel width is 125 kHz. According to [8], the low data rate optimization bit is enabled whenever $T_{sym} > 16$ ms. Thus $DE = 1$ only for $SF = 12$ and $SF = 11$, $DE = 0$ otherwise. With this Using eq. (3.11) the total airtime then becomes:

$$T_{uplink} = T_{preamble} + (n_{payload} \times T_{sym}) \quad (3.12)$$

LoRaWAN data rates

There now 7 defined data-rates(DR) in the LoRaWAN-specification[4], these are summarized in table 3.3. The data-rate varies with the bandwidth(BW) and spreading-factor(SF). The higher the BW and lower the SF, the higher the data-rate. This is evident by looking at the airtime calculation in section 3.2.6. All devices supports BW of 125 kHz, larger BW is not as commonly supported.

Data-Rate(DR)	Spreading-factor/Bandwidth	Physical bit rate
0	SF12 / 125 kHz	250 bit/s
1	SF11 / 125 kHz	440 bit/s
2	SF10 / 125 kHz	980 bit/s
3	SF9 / 125 kHz	1760 bit/s
4	SF8 / 125 kHz	3125 bit/s
5	SF7 / 125 kHz	5470 bit/s
6	SF7 / 250 kHz	11 000 bit/s
7	FSK(frequency shifting keying)	50 000 bit/s

Table 3.3: Supported data-rates in LoRaWAN[4].

3.3 MQTT

MQTT(MQ Telemetry Transport) is a publish/subscribe messaging protocol. It consists of 2 entities: a client and a broker. The client subscribes on something called topics which the broker provides. The clients can publish messages on the topic they have subscribed to. Every time the broker receive a message on a certain topic, it notifies(publishes) said message to every client that subscribes to that topic. In other words, the broker routes messages to and from different clients. An example of how the protocol works is given in fig. 3.7. In the illustration, all the clients subscribes to the same topic.

3.4 NTP

NTP(network time protocol) is a networking protocol for clock synchronization between computer systems. It is made specifically for systems working on a packet switched network, like Ethernet. The protocol is made to synchronize network-entities within a few milliseconds to coordinated universal time(UTC). One way to synchronize with the

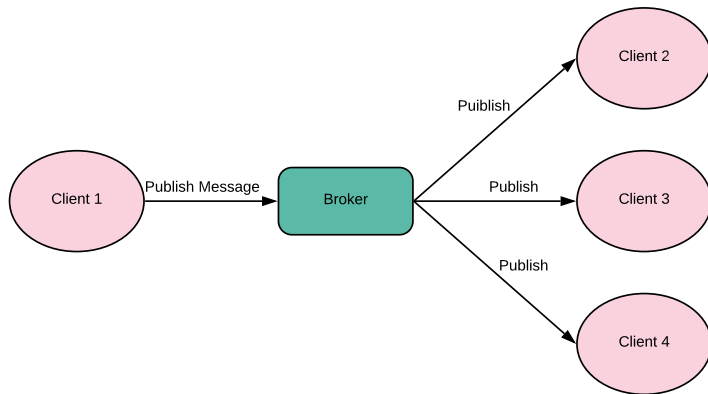


Figure 3.7: Collaboration diagram illustrating the interaction between a broker and its clients.

time server is to send a NTP request via UDP on port 123. The corresponding answer will contain the current time. There are several public time-servers available, the one used in this thesis is `no.pool.ntp.org`.

Chapter 4

Specification and Design

The chapter is divided into four main parts, covering in order: The functional specification, Technical specification, Acceptance criteria and design. For each of these sections, the requirements for each of the main parts of the system will be derived.

4.1 Overview

This first section will focus on giving an overview of the system which consists of An end-device, Gateway and an RT-server/back-end. A system overview is provided by the collaboration diagram in fig. 4.1, this shows how the different entities are connected.

The embedded device(End device/Node) awakes by being triggered by an event, upon awaking the device will store vibration-data. Then transmit said data via LoRa to a gateway. Upon receiving the message(uplink) from the end device, the gateway will wrap the message data into a JSON object and forward it to a handler/server. The server will then store the vibration-data. If necessary the server will produce a callback-message that is being sent back down to the end-device. The callback can contain configuration-data for the end-device. The callback is sent to the gateway and gets processed to generate a downlink. The downlink can be stored in a downlink-queue on the gateway, if the downlink window has passed. If the downlink window haven't passed, the message will be aired immediately if the downlink window is still open.

The complete system will provide various timestamped vibration-data from a number of end-devices deployed in an area of LoRa coverage. This will help differentiate between messages. Each end-device must be calibrated before they are deployed.

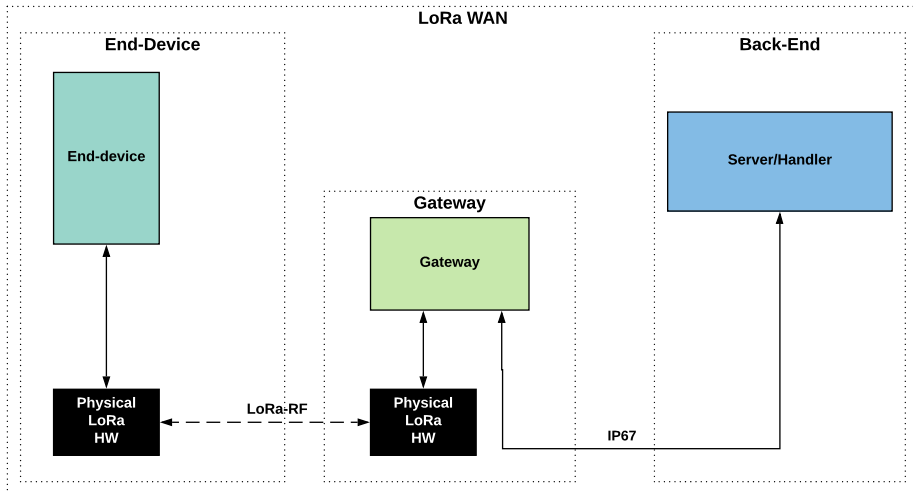


Figure 4.1: System collaboration diagram/overview.

4.2 Functional requirements

As a remote vibration sensor, the end-device must be portable and use LoRa-RF to communicate with the server. Furthermore the end-device must be able to measure vibrations which in turn are processed and sent to a back-end/server. The measured values must be timestamped in order to be aligned during analysis. The device must be battery-driven and in turn rechargeable. To ensure future re-use it should also feature interface for reprogramming and peripherals. Furthermore, it should also feature human interfaces such as status indication and buttons. The server must be able to store data and generate callbacks. Furthermore, it should have a clock with which a end-device can synchronize. Upon storing the data, the data should be timestamped.

1. End devices should be fully portable and able to collect vibration data in addition to various internal diagnostics.
2. The system must provide 2-way communication to be able to configure devices remotely.
3. To provide reliable time-stamping of vibration data, the end devices and server must be synchronized.
4. The data sent from devices must, in turn, be stored on some kind of server for analysis.
5. The device should have a long battery life(months).
6. The device must have interfaces for both debugging and peripherals.
7. Each device's sensor should be calibrated.

Table 4.1: Functional specification for the system.

4.3 Specification

This section states the technical specification for the system. It covers all three parts of the system in order: end-device, gateway, and back-end/server.

4.3.1 End-device

To easier track the requirements during development, they are split into two different context's: embedded hardware and embedded software. This is a more in-depth analysis of the previously stated functional specification given the previous section and summarized in table 4.1.

Embedded hardware

Based upon the requirements stated in section 4.2, the technical specification for the node is stated in table 4.2

1. Must be battery-driven.
2. The end-device should be rechargeable.
3. Should be powered during charging.
4. External charging power should be provided by 5V USB, which is a readily available interface.
5. Low-power as it is battery-driven.
6. LoRa-capabilities(LoRa RF-module).
7. Ability to measure vibrations(accelerometer).
8. Battery diagnostic(level indication).
9. Interface for debugging, programming and peripherals.
10. Interface for reset and sending dummy messages.
11. Visual status indication.

Table 4.2: HW-specification.

The above stated specifications should in-part cover Pt. 1, 5 and 6 stated in table 4.1.

Embedded software

The software must collect data from different peripherals(gyro and luminosity) and inner-diagnostics(battery level). These data must be forwarded to a back-end/server via a LoRa RF-transmitter. And must be done in an energy-efficient manner to preserve battery life. Keeping track of when such events occur the dis done by implementing an RTC with timestamps. A summary of the specification is provided in table 4.3.

1. Sleep modes should be used to save energy.
2. Should provide an interface for LoRa uplinks and downlinks.
3. Must be able to detect and store vibration data.
4. Must be able to send stored vibration data to the server.
5. Provide an RTC for time-stamping events.
6. Be able to synchronize with a backend/server.
7. Be able to detect external events caused by vibrations.
8. Have a routine for accelerometer-calibration.
9. Use LoRa to receive configuration parameters.
10. Handle unexpected errors.
11. Have one or several interfaces for debug-purposes.

Table 4.3: SW-specification.

The above stated specifications should in-part cover Pt. 2, 3, 4, 5 and 6 stated in table 4.1.

4.3.2 Gateway

The gateway should comply with the regional parameters in EU863-870 MHz ISM Band[4]. It should also work as a simple package-router with minimal processing time, in this way keeping the GW-processing overhead to a minimum. This is essential to be able to propagate the message to the server, and in turn, receive a potential callback before the downlink window has passed. If the downlink-window passes, synchronization is not possible. The requirements are summarized in table 4.4.

1. Must handle uplinks and downlinks before a downlink-window passes.

Table 4.4: Gateway-specification.

The above stated specifications should in-part cover Pt. 2 and 3 stated in table 4.1.

4.3.3 Server/Handler

The server side of the system must keep a global time, with which any end-device can synchronize with. In addition to keeping the time, the server should be made specifically for real-time. The server should also be able to get the current POSIX-epoch from a time server, this is to be synchronized(within some seconds accuracy) with the current date and time. It must be able to communicate via IP-67(Ethernet, UDP or TCP) with the GW. The server should have timers that can be used to calculate RTT(round trip time) and callback time. Uplinks from end-devices should be answered with a callback when required. The server must be able to log uplink payload-data to file for later analysis.

1. Must be able to synchronize with a time server.
2. Must be able to send a callback.
3. Must be able to log payload-data with the corresponding timestamp.
4. Should be able to synchronize with an end-device by sending its current timestamp.
5. Must be able to send a callback/downlink to an end-device.
6. Must have either SW or HW-timers.

Table 4.5: Server-specification.

The above stated specifications should in-part cover Pt. 2, 3 and 4 stated in table 4.1.

4.4 Acceptance criteria

The acceptance criteria was derived from the requirements stated table 4.2, table 4.3, table 4.4 and table 4.5. The system will work as intended if it passes all the criteria stated in table 4.6.

Label	Description
AC1	Vibration data and device diagnostics sent from an end-device are stored.
AC2	The server stores the time-stamped vibration-data in order.
AC3	End-devices are calibrated.
AC4	End-devices synchronizes with the server in order to have a common clock.
AC5	The server is synchronized with a time-server.
AC6	End-devices receives and applies configuration parameters from the server.
AC7	End-devices handles unexpected errors.
AC8	End-devices has a long battery-life and can run for months without recharging.
AC9	The end-device enters sleep mode when idle.
AC10	When an end-device dies, it is detectable at the back-end.
AC11	End-devices can be placed wherever as long as a LoRa-gateway is reachable.
AC12	End-devices prints debug-data to terminal.
AC13	End-devices has status LED's indicating the current status.
AC14	The end-device has interface for peripherals with I2C, UART, SPI.

Table 4.6: Acceptance criteria for the complete system.

4.5 Design

This section covers the system design. The design for each part of the system will be provided here, except the gateway. Since the gateway is a simple packet/message forwarder, there are no underlying design-features to be uncovered. Thus this is skipped in this section.

4.5.1 PCB design

The PCB design was based upon the specifications given in table 4.2 and resulted in the implementation seen in fig. 4.2.

As the diagram shows the PCB is driven by a battery that can be recharged. Thus making it a completely wireless device. As it is battery-driven an LDO Regulator is suitable to regulate the power level from the battery. The battery circuit also features 2 status LEDs for power status and charge status. Furthermore, the board contains a LoRa RF transmitter, which is compliant with both LoRa Class A and B. The accelerometer comes in the form of

a mems-sensor, it provides an accelerometer, gyro, and package-thermometer. 2 buttons are added, one for sending a dummy-message, and one for hard-resetting the device. To indicate current MCU-status, an LED array is added, which contains 3 LEDs of different colors. Lastly, IO-headers are provided for interfacing external devices with the various buses on the MCU.

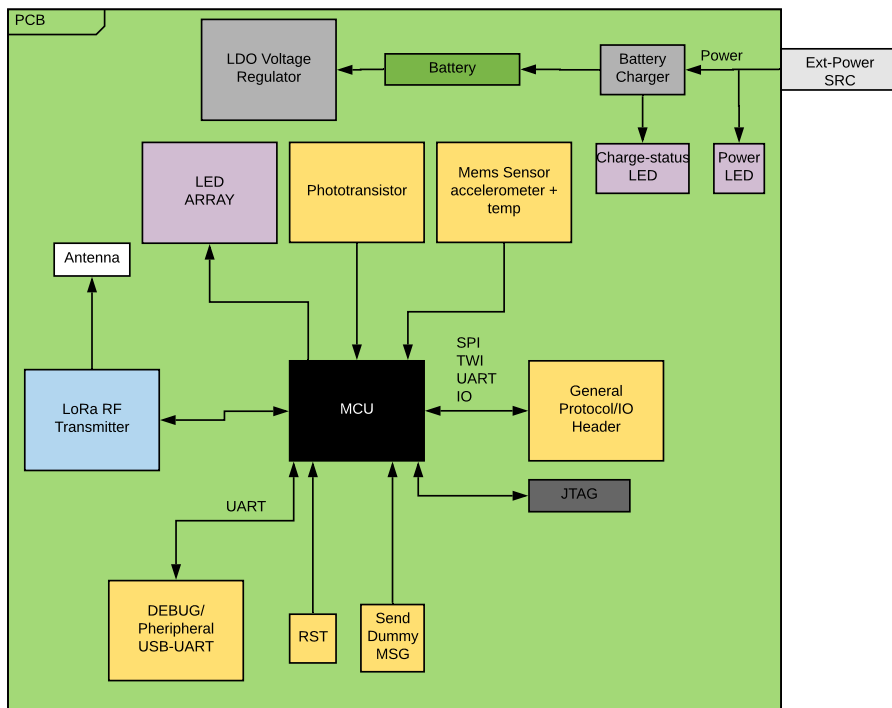


Figure 4.2: Block diagram of the PCB.

4.5.2 Back end

The back-end of the system has 2 configurations: Using an RT-server or using The things network[10] as a service provider. This section will cover both of these configurations.

RT-server

For the sake of investigating over the air synchronization, the intent is to have as low overhead as possible. Furthermore using IO pins to check time-stamping makes the use of

a normal computer difficult. Therefore the server/back end must be RT-compatible device, preferably with GPIO and IP67 compatibility. This is to both communicate with a private GW and to synchronize with a time-server using NTP(net time protocol). Combining this with a configurable LoRa gateway should be sufficient to conduct a somewhat accurate investigation of synchronization accuracy. The following diagram illustrates the back-end of this configuration.

The things network

Using a single gateway is somewhat limited because the high-power gateways available aren't mobile. Thus to increase the overall coverage of the network, it was decided to use the already available network provided by The thing network(TTN). Doing this makes it easier to deploy nodes in a possible field test. However it limits the customizability of the gateway and server, however coverage trumps this. The diagram in fig. 4.4 illustrates the back-end when using TTN. TTN provides a API with which developers can implement a MQTT-client, to which TTN's handler forwards uplinks-messages. This means that the only thing to configure is the TTN broker and a custom client.

4.5.3 System design

The full system design can be illustrated in two interface diagrams, one for each of the back-end configurations. The first one with the RT-server is used for investigating synchronization. The second using TTN is used for field deployment. Both is illustrated in fig. 4.3 and fig. 4.4 respectively.

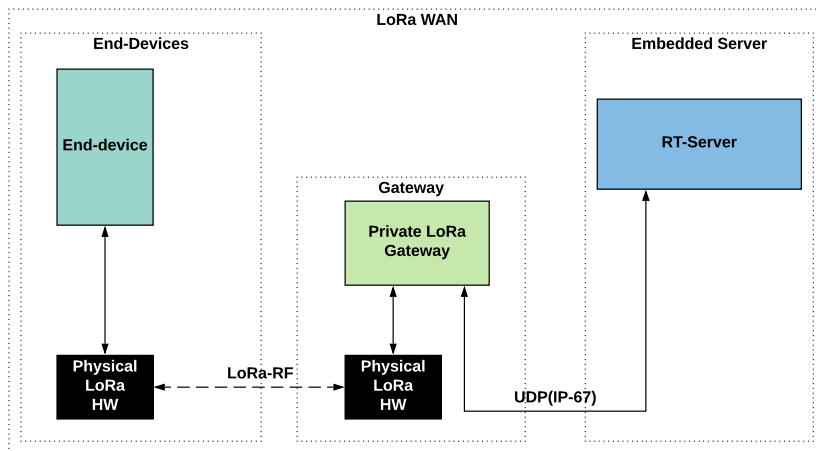


Figure 4.3: System design utilizing an RT-server.

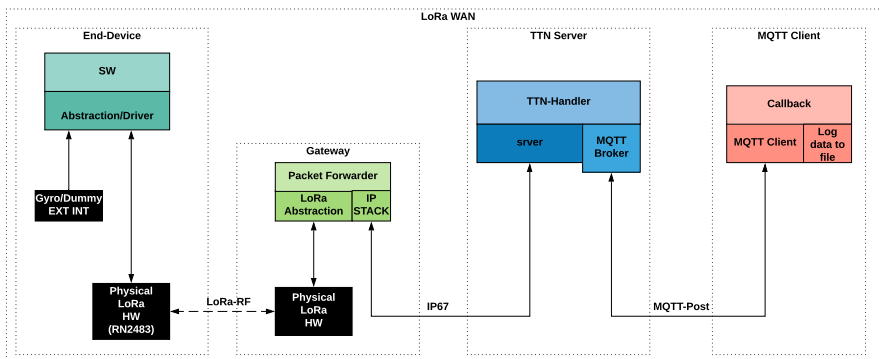


Figure 4.4: System design when using The things network[10]

Chapter 5

Implementation

This chapter provides details on how the different parts of the system have been implemented. As stated in 4 it consists of three main parts: End-device, Gateway, and Back-end. The back-end comes in two configurations, thus these are covered in turn by two sections. Furthermore, the implementation of the end-device has been split into 2 parts: Hardware and software.

The main tool for developing schematics and PCB was *Altium Designer*. The PCB is a 2 layer design which features components on both the top and bottom side for a more compact and space-efficient design. Most of the components are surface mounted as these require less space and fewer drill holes.

5.1 Hardware End-device

The full schematic for the end-device can be found in Appendix B1, with corresponding part-list given in Appendix B2. The parts have been selected to create a generic embedded-device, with some considerations taken to make it somewhat energy efficient. The scope is to investigate the use of LoRa in a distributed embedded system, in other words: a battery-driven vibration-sensor with LoRa capabilities.

5.1.1 Components

This section provides an overview of the components necessary to realize the schematic seen in Appendix B1.

MCU Atmega324PB

The Atmega324PB was chosen because it is a known system for the developer. After all, the same MCU was used in an earlier project[9]. The thought is that both tools and hardware is easier to navigate. The Atmega324PB micro-controller is a 8 bit high-performance processor utilizing the AVR-based RISC architecture. It comes with 32 kB Program memory, 1 kB EEPROM storage and 2 kB SRAM. More importantly, it features three separate UART buses, two separate I2C buses, SPI-bus and GPIO. Making it quite versatile when interfacing with various peripherals. It also features an ADC(Analog to Digital Converter), and both 8-bit and 16-bit timers/counters in addition to a real-time counter with an optional separate crystal. As most of the micro-controllers in the AVR-family, the Atmega324PB also features an IEEE compliant JTAG interface, which can be used for both debug and reprogramming. For clock source, the Atmega324PB either uses an 8 MHz internal crystal or an optional 1-20 MHz external crystal. It was decided to use a 1.8432 MHz crystal as a clock source. This was because this frequency gives an error of 0.0 % on the UART module on the bit-rates $2.4\text{-}57.6 \frac{\text{bit}}{\text{s}}$. The alternative was to choose a 1 MHz crystal, even though this yields slightly less energy consumption, it also provides an error of up to 8% on the UART.

The real-time counter is ideal for implementing an RTC, as it can be fitted with a 32.768 kHz crystal. Based on the specification in section 4.3, the Atmega324PB is a good fit.

LoRa transceiver RN2483

The RN2483 is a low-power LoRa transceiver developed by Microchip in accordance with LoRaWAN Class A protocol. It integrates RF, base band controller and a command API processor. Which makes it suitable to interface with an external host MCU, where the host MCU can send API-commands directly via UART. For simplicity the transceiver features a fully implemented protocol stack for LoRa class A [11]. It requires a 2.1-3.6 V input voltage, and can go into a sleep mode consuming only 2-26 μA [12] depending on the actual input voltage.

Battery Samsung INR18650-25R

It was decided to use a single cell 18650 Li-ion battery as power source. These cells are one of the most used in the industry today, their application ranges from e-cigarettes, power-banks to laptop-batteries[13]. The Samsung INR18650-25R is a 18650-sized single cell Li-Ion battery, providing a nominal voltage of 3.6 V and a cut-off at 2.5 V[14].

Battery charge controller - MCP73830

The MCP73830 developed by Microchip is a single-cell Li-Ion charge management controller. It requires a low number of external components when implemented, and power can be drawn from the battery whilst charging. This fits well with the specifications set in section 4.3. With a input voltage range from 3.75-6 V, it interfaces well with the well known USB-standard for external power. It can provide a 20-1000 mA[15] charging current at 4.2 V, which is the recommended charge rate for the INR18650-25R[14]. Upon implementing the charge controller it was decided to go for a 500 mA charging current.

Accelerometer MPU-6050

The MPU-6050 developed by InvenSense is a 6-axis Motion-tracking device with 15-bit resolution. It features a gyroscope, accelerometer and thermometer. The each data sample is stored in 2 8-bit registers, and is read like a 16-bit signed integer. The accelerometer has 4 levels of sensitivity: $\pm 2g$, $\pm 4g$, $\pm 8g$ and $\pm 16g$. In addition to these sensors, it comes with a built in 1024 B FIFO buffer, which can store sampled data. A nominal input voltage between 2.4-3.4 V is required. The possibility to go into a low-power mode at 140-10 μ A[16] makes it favorable in a battery-driven device. It also features an external interrupt-pin, which can be triggered by either a motion-interrupt, FIFO-overflow or data-ready. I2C provides the main interface with a external host MCU and other peripheral devices.

LDO voltage regulator NCP718

The NCP718 from ON Semiconductor is a 300 mA low drop-out voltage regulator. It comes with an ultra-low quiescent current[17]. It has a input voltage range of 2.5 V to 24 V and a fixed output of 1.2-5 V. Based upon the cut-off voltage of the chosen battery

in section 5.1.1, this makes for an extra safety barrier protecting the battery. An output of 2.5 V would be ideal since this utilizes the most of the battery capacity before shutting of. This and the ultra-low quiescent current makes it a good fit for the end-device, according to the specifications given in section 4.3. It also fits all the required input voltages for the other components mentioned earlier in this section.

5.1.2 Power circuit

The schematic of the power circuit is shown in fig. 5.1. As seen the power is taken from the connector J2, which is a micro-USB type connector. This provide 5 V power to the charging controller MCP73830(U4). The charging current is configured by the resistance in R1, the current is set according to the following formula found in the data sheet[15]:

$$I_{charge} = \frac{1000}{R1} \quad (5.1)$$

Where $I = \mu\text{A}$ and $R = \text{k}\Omega$. As seen in the schematic R1 is 2 k Ω , which according to eq. (5.1) gives a charge current of 500 mA. Furthermore there are 2 LED's, D1 and D2. D2 will light up as long as there is power from J2, whereas D1 will give feedback of the charging status. It will blink according to the output states given in Table 5-1 in the charge-controllers data-sheet[15]. The charge-controller provides power to the 18650-Li-Ion battery, this is held in place with simple clam-mountings. The output of the battery provides power to the LDO voltage regulator U5. The two capacitors C1, C3 and C4 provides a low impedance path to help reduce any noise.

Battery level indicator

As stated in section 4.3 the device should be able to read it's battery status. The simplest way to do this is to use a ADC to read the battery-voltage directly. However the battery voltage varies from 4.2-2.5 V whereas the Atmega324PB operates on 2.5 V. The solution was to create a voltage divider to scale down the battery voltage from 4.2 V to 2.5 V. This can be seen in the schematic by looking at R8 and R7, with the outlet for the ADC V_{bat} in between. By using the following formula:

$$V_{bat} = V_{source} \times \left(1 - \frac{R8}{R8 + R7}\right) \quad (5.2)$$

Where V_{source} is the battery output voltage and V_{bat} goes to the ADC input. The values of R7 and R8 can be calculated, however these vary depending on the current through R8 and R7. Table 32-2 in the Atmega324PB data-sheet[18] states that the lowest pin input leakage current is $1\ \mu\text{A}$. Thus the maximum total resistance can be $4.2\ \text{M}\Omega$. However to ensure that the battery will be read, it was decided to go a total resistance of $420\ \text{k}\Omega$, therefore due to limitations in available resistances in real life, the R7 and R8 was chosen to be $255\ \text{k}\Omega$ and $174\ \text{k}\Omega$ respectively.

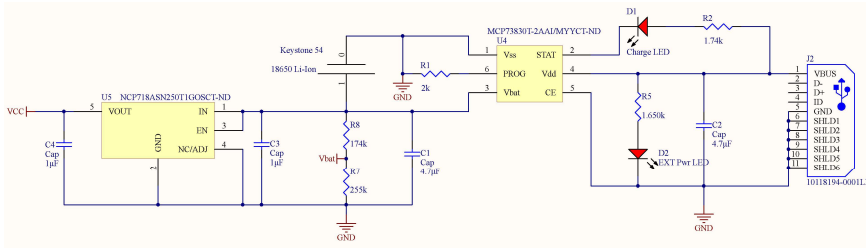


Figure 5.1: Schematic presenting the power circuit of the end device.

5.1.3 Accelerometer circuit

The schematic seen in fig. 5.2 presents the implemented circuit of the MPU6050 as U2A. According to section 7.2 in the data-sheet[16] it is recommended to have 2 bypass capacitors on VLOGIC an VDD, C12 and C13 were therefore added. Furthermore as earlier mentioned the MPU6050 features an external interrupt. This is interfaced with a pin on the Atmega324PB, to notify the MCU upon an event.

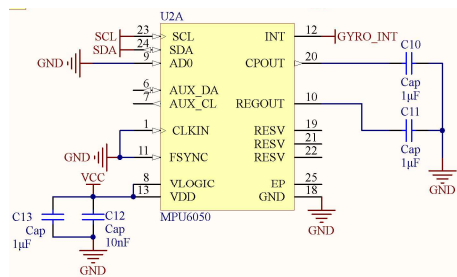


Figure 5.2: Schematic presenting the circuit of the accelerometer.

5.1.4 RF transmitter circuit

The schematic in fig. 5.3 show the implementation of the RN2483(U3). As seen in the schematic, it is interfaced with the UART0 on the Atmega324PB and interconnected with a global RST. The header J2 is a ICPS header, it was implemented for flashing new software to the RN2483. The 2-radio outlets(RFH and RFL) are connected to a common microcoaxial RF-connector(MCRF) to which a antenna can be attached.

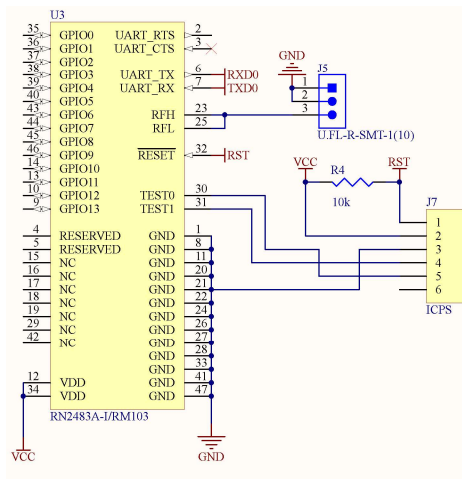


Figure 5.3: Schematic presenting the circuit of the LoRa transmitter.

5.1.5 Headers and peripherals

As seen in the schematic found in Appendix B1, three LED's(D3, D4 and D5) were added to give some simple feedback to the user. D3, D4 and D5 emit green, yellow and red light respectively when toggled.

The photo-transistor Q1 were added to collect some extra information from the nodes. It reacts to ambient light in the spectral range of 300 nm-950 nm.

The buttons S1 and S2 are for hard-reset and to trigger an external interrupt. S1 pulls the RST-line to ground, causing a reset on both the MCU and the RF-transmitter. S2 is connected directly to an external-interrupt(INT1) pin on the Atmega324PB, which when triggered sends a dummy-message.

All the free pins on the Atmega325PB is connected to corresponding headers This is to ensure an interface with which any optional external devices can connect to. The headers J6

and J11 was added to "sniff" on the communication with the MPU6050 and RF-transmitter. J11 was solely added for debug purposes. However J6 is an I2C-line and can be used as an interface with other slave-devices with an I2C interface.

5.2 PCB result

The PCB created and designed in *Altium Designer* and a prototype board were ordered from *JLPCB* in China. This was a 2-layer PCB, with a power-plane on top and ground-plane on the bottom. A section of the top-plane were separated as a 5 V power-plane for the charge-controller. The PCB trace, without the GND-plane or VCC-plane, can be seen in Appendix B3. Upon arrival the prototype was soldered and tested at NTNU. After testing had conceded, a batch of 15 fully assembled devices were ordered from the Chinese site *PCBWAY*. The final board dimensions were 79.9 mm x 49.6 mm, and can be seen in fig. 5.4.



Figure 5.4: The assembled End-device with battery attached.

5.3 Software End-device

The software developed for the Atmega324PB was entirely developed from the ground up. Except for the MPU6050 driver and I2C driver. These drivers were found online and heavily modified, the original drivers are open-source and can be found on *GitHub*[19]. The software modules are described in section 5.3.4. Some software were imported from an earlier project[9], however they were somewhat modified.

The main tool for developing the software was *Atmel Studio 7.0*, this is a complete tool which includes device-programming(setting fuses) and a functional debug tool. It was mostly written in C++ using the AVR-G++ toolchain. Some drivers were written in C as this was deemed more practical. The device was flashed using the *Atmel-ICE* programmer. The drivers has been thoroughly tested, however the main program still possesses some minor bugs.

The software-development were documented using *GitHub*¹. The software itself have *Doxygen* comments. *Doxygen* generates documentation based on comments in the code and can be exported to HTML or PDF.

5.3.1 RTC

The end-device must be able to timestamp events, thus it must have a clock which keeps the local time of the node. The solution was to use the 8-bit RT-timer(Timer 2) on the Atmega324PB, this can be implemented with an external 32.768 kHz crystal. The big advantage with this frequency is that it is equivalent to 2^{15} , which makes it perfectly divisible with 2. Thus on the 8-bit timer with the prescaler set to 64, the timer will overflow 2 times per second. Meaning that every other time the overflow interrupt is triggered a second has passed. The ISR(interrupt service routine) will then increment a 32-bit unsigned integer which is used to store the local time. With a 32-bit integer the device can be synced using *Unix* time, however the resolution of the timer is still only 1 s or at best 0.5 s.

It was thought beneficial to increase the resolution by using a different timer to keep track of the μ s and ms. Thus another 8-bit timer(Timer 1) was integrated into the RTC, this uses the 1.8432 MHz crystal as a source. With no prescaler the ticks per ms can be calculated

¹Repository available at https://github.com/tobulf/Master2019_End-device

as:

$$0.001 \text{ s} \times 1843200 = 1843.2 \quad (5.3)$$

and

$$\frac{255 \text{ ticks}}{1843200} \approx 139 \mu\text{s} \quad (5.4)$$

Which equals to an timer overflow of 7 and a reminder of 51.2. Thus for every timer overflow, approximately 139 μs has passed. Additionally for every 7th overflow, a ms has passed, and for every 52nd ms the clock must be corrected by subtracting 1 ms. Both the microseconds and milliseconds are stored in 16-bit unsigned integers.

By using Timer 2 to keep track of the seconds and Timer 1 to keep track of the ms and μs , the RTC has a resolution of $\pm 139 \mu\text{s}$. It is worth mentioning that Timer 2 resets Timer 1 for every second that passes, thus keeping them synchronized. This also means that when the ISR of timer 2 and 1 is triggered, all other interrupts are disabled until the ISR has finished. This is to ensure atomic access which prevents race-conditions.

Both timers were wrapped into an object and interfaces to set at get time was implemented. Using objects ensures that the user can't abuse the driver by accessing variables directly. An alarm system was also implemented, in which a user can set a periodic alarm. This can be used to wake up the device periodically when in sleep mode.

5.3.2 Timer

A useful tool when investigating timing in embedded HW is timers. Thus it was decided to implement a high resolution timer-object. This was done by using the three 16-bit timers on the Atmega324PB, Timer 1, Timer 3 and Timer 4. The timer object was implemented in such a way that every declaration of a object starts a new timer. This restricts the number of declarations to 3, if more objects are declared the consequent objects will return a zero value. The destructor ensures that the timer is reset and consequently turned off. Thus timers which are not in use won't be running. There are 4-interfaces, for resetting, stop, start, read μs or read ms.

5.3.3 Watchdog timer

The Atmega324PB has a built in watchdog timer, which can be configured to trigger a reset, ISR or both ISR and reset upon overflow. It utilizes a dedicated 128 kHz oscillator and a 2K - 1024K prescaler which gives a timeout between 16 ms to 8 s. However since the WDT(Watchdog timer) can be configured to trigger an ISR, it can be implemented to trigger a reset on a multiple of the given timeouts. This can be done by increasing an integer every time the ISR is triggered, after a certain amount of timeouts, the ISR sets the WDT to reset upon the next timeout.

The main usage of the WDT is to prevent deadlock or starvation. This can happen in routines which polls a signal that never change or awaiting an answer which never come. With a WDT the device will be stuck in these situation for only a certain amount of time, before it restarts.

5.3.4 Drivers

UART

There are three separate UART hardware modules on the Atmega324PB, two of which are in use on the end-device. These are used to interface with the radio and debug/printing to terminal.

The first module (UART0) is interfaced with the radio-transmitter to send and receive commands. This is done by a dedicated object(LoRa_COM) which works as a driver for this module. This enables the developer to not only send and receive single characters and strings, but also to send break conditions. When waiting for an incoming string, the MCU is set to idle mode with the UART receive interrupt enabled, upon receiving a byte it will wake up. This is to reduce the polling-time and energy consumption.

The second module UART2, is written in C and imported using the macro *EXTERN*. This enables the C library function *printf*, can write to terminal via the UART. This is handy for debug-purposes during development.

As mentioned the Atmega324PB has a third module(UART1), however this cannot be used when it shares a pin with the external interrupt(INT0) which is connected to the MPU6050.

RN2483 - Radio driver

The RN2483 driver inherits the functionality from the UART-driver `LoRa_COM`. Furthermore, it features functions for OTAA join procedures. Setting data-rates, receive window sizes and channel duty cycles. Interfaces for sending and receiving LoRa messages on different ports. Procedures for putting the RN2483 into sleep mode and waking it. Upon sending a message the object returns a Boolean value depending on the success. However, upon receiving a message, the object stores the payload in a buffer and sets an internal Boolean value. To consume the value, the developer can check if there are messages present, and then if so consume the message.

It is worth mentioning that in some cases a confirmed message can take over 8 s to be successfully sent. This depends on how the radio is configured, however, this means that the WDT must be able to have a timeout which is more than 8 s long.

For simplicity it was decided to import the *String* data-type from the official *Arduino* library[20]. This simplifies the handling of strings and has almost no extra overhead.

EEPROM

The Atmega324PB has 1 kB of EEPROM available. A simple EEPROM driver was written, to store MPU6050-accelerometer offsets after calibration. It features functions to write and read single-bytes, which in turn are used in functions to read and write signed 16-bit integers. Attached to the EEPROM driver, is a header file that contains the defined memory locations for the different types of data.

MPU6050 - Accelerometer

As earlier mentioned the driver for the MPU6050 was imported together with an I2C-driver. However to suit the application it the MPU6050 driver was heavily modified. There was functionality for both initialization and reading measured values. The latest measured 16-bit data are stored in 2 1-byte registers on the MPU6050[21]. Each of the six-axis has its register-space for storing the latest data. These values are raw data, such that they must be converted into the software using the appropriate gain and offset. The gain values are given in the MPU6050 data-sheet[16]. The initialization procedure was modified such that upon initializing(if calibrated), the different calibrated-offsets are read from the EEPROM.

The MPU6050 can be set in a low-power mode[21], this is done by enabling sleep mode and setting a wake-up period. Upon waking up, the MPU6050 takes 1 sample, checks for interrupts then goes back to sleep. This can be combined with disabling the gyro and temperature, to lower the energy consumption to a minimum. Functions to switch between normal and low power mode where thus added.

The MPU6050 also features an external interrupt-pin, this is connected to an interrupt pin(INT0) on the MCU. The MPU6050 has 3 interrupt-modes[21][22]: FIFO overflow, motion-threshold and data-ready. To check which interrupt is triggered, the MCU has to read the interrupt status register on the MPU6050. The threshold for motion-interrupt can be set by writing to a register on the MPU6050. This was not included in the original driver and therefore added. The interrupt is mainly used to awake the MCU upon a sudden motion surge and FIFO-overflow.

A 1024 B FIFO-buffer[16] is available to store the measured data on the MPU6050. When enabled, every time there is new data available, these are put into the buffer in order. This means that the 1st and 2nd byte in the FIFO buffer corresponds to the acceleration on the X-axis, the 3rd and 4th to the Y-axis and the 5th and 6th to the Z-axis[21]. This changes if the gyro and temperature sensor is enabled, however, in this case, these disabled. The FIFO buffer can be used to take bursts of samples or store data over time. When the buffer is full, and a new value is written, the FIFO-overflow interrupt(if enabled) is triggered. The buffer can both be written to and read from. An additional register stores the current length of the buffer, which is updated every time the buffer is either read from or written to. The buffer is mainly used to store and read measured samples from the MPU6050-accelerometer.

ADC

A 10-bit ADC with 7 channels is embedded in the Atmega324PB. A simple driver was written to read the battery and light-sensor voltage level. It features functions to get the light and battery levels in percent. It can also be enabled/disabled when not in use. The ADC was calibrated using a *GwINSTEK* GPS-3030 voltage supply.

LED's

The status LEDs driver is a simple driver, which features functionality for toggling, resetting and turning on the different LED's. This control whether or not the green, yellow or red

status-led is on or off.

5.3.5 Accelerometer and temperature calibration

To provide accurate measurements, each accelerometer(MPU6050) has to be calibrated individually. A flowchart of the following explained scheme can be seen in fig. 5.5 Therefore a simple calibration procedure was implemented.

The procedure is simple: The device is put on a sturdy surface with has little to no vibrations, this can be a concrete floor, wall or roof. The accelerometer is then set to a 20 Hz sample rate, $\pm 2G$ sensitivity, and the FIFO buffer are enabled together with FIFO overflow interrupt. When the interrupt is triggered, the accelerometer has taken 1024 B of sample data. Given that the accelerometer measures on all three axis(X,Y,Z), this results in: $\frac{1024 \text{ B}}{3 \times 2 \text{ B} \times 20 \text{ Hz}} \approx 8.5 \text{ s}$ of sampled data. Since the buffer can't store an even amount of data from all three axes, it is easier to read only the first 8.5 s of data, which is represented by the first 1020 B in the buffer. When the FIFO overflow interrupt is triggered, the MCU disables the FIFO buffer, such that no more data is written. The MCU proceeds to calculate the mean value for each axis by summarizing each sample according to the axis before dividing by the total sample number.

The temperature sensor is calibrated by taking 1 single raw sample, then searching for the offset by adding/subtracting the gain to/from the raw sample before converting it. The converted is given by: $\frac{T_{raw} - T_{offset}}{T_{gain}} = T_{converted}$.

Upon calculating the mean values and finding the temperature offset, the MCU stores each value in the device's EEPROM. These offsets can be used for every other sensitivity as well. This is done is by applying a fraction of the calculated offset as the current offset. As the $\pm 2G$ sensitivity is half that of the $\pm 4G$, the offset should also be half as big in comparison. This approach is used when changing the sensitivity from the to anything apart from $\pm 2G$.

5.3.6 Collecting vibration data

The data vibration-data collection is done mostly by the MPU6050(Accelerometer). When configured in low-power mode and with the motion-interrupt enabled. The MPU6050 will provide an external interrupt by setting an external INT-pin to low. This will happen whenever a large enough motion is detected. This will in turn trigger an ISR on the

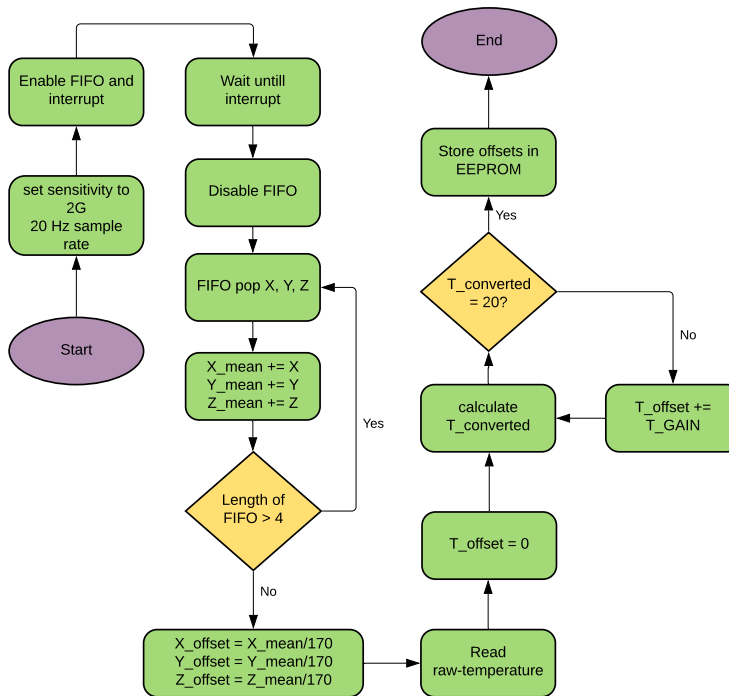


Figure 5.5: Flowchart illustrating the calibration scheme.

Atmega324PB. The workflow of the MCU-ISR is illustrated in fig. 5.6. Upon sensing an MPU6050 interrupt, the MCU immediately read the interrupt register of the MPU6050 and disable the MCU sleep-mode. If it is a motion interrupt, the timestamp is saved in a global variable and the MPU6050 is set into normal power mode. The FIFO-buffer is then activated together with the FIFO-overflow-interrupt on the MPU6050. This means that the next time the interrupt is triggered, it is because of the FIFO buffer overflowing. While the FIFO fills up, the MCU will continue to sleep or do whatever it did before. When the FIFO is full, the ISR is triggered once again. The FIFO will then be disabled(not emptied) together with the MPU6050 interrupt. The interrupt remains disabled until the data in the FIFO-buffer has been consumed and sent. An global Boolean variable is set true in order to notify the IDLE state(see fig. 5.9) that there is data available.

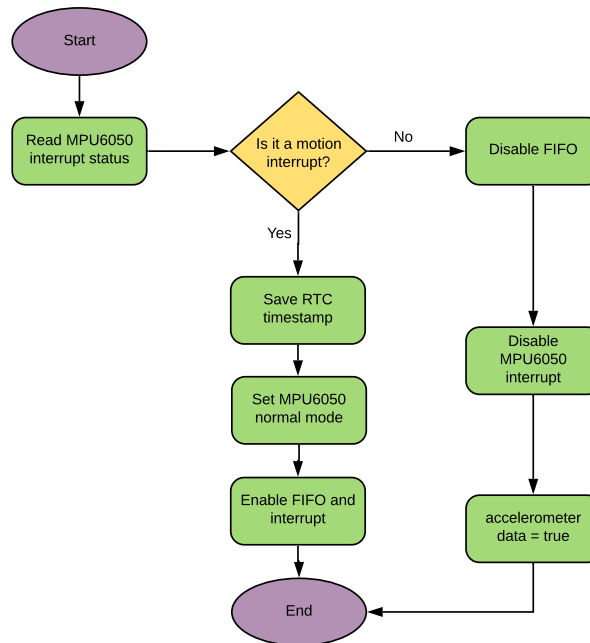


Figure 5.6: Flowchart illustrating the operations of the MPU6050 ISR.

5.3.7 RTC alarm

The RTC contains as earlier mentioned an alarm. This alarm can be set to trigger periodically within a second accuracy. The way it works is that every other time the Timer 2(RTC) overflow ISR is triggered, the MCU checks if the difference between the last triggered alarm and the current time is bigger than the configured period. If this is the case, the RTC disables sleep-mode and sets an internal Boolean variable to true and set up the next alarm. The sleep-mode is disabled to wake the MCU from sleep if it is in sleep mode at the time of the alarm. The flowchart in fig. 5.7 illustrates the workflow of Timer 2(RTC) ISR. The Boolean value is reset whenever the `RTC.get_alarm_status()` is called. This is done whenever the state machine is in the IDLE state.

5.3.8 Main program

In normal operation mode, the device act in accordance to the state diagram in fig. 5.8. The End device has 4-states: IDLE, SLEEP, DATA TRANSMIT, and ALIVE TRANSMIT.

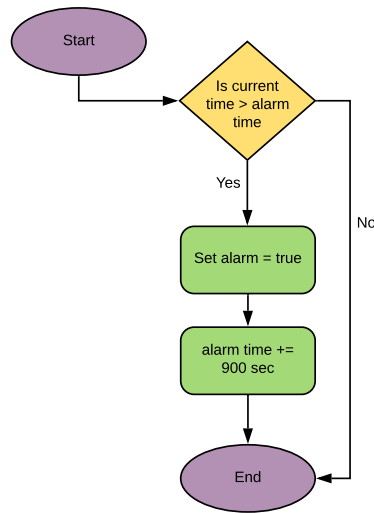


Figure 5.7: Flowchart illustrating the operations in the RTC ISR(Here the period is 900 s / 15 min).

Unless there is an RTC-alarm, an accelerometer-event or the message button is pressed. The following sections explain each state. It's worth noting that synchronization is not implemented in this device, this is further explored in section 5.3.9.

SLEEP state

The Atmega324PB has several power management modes. These can be used to preserve energy in situations when the MCU and its peripherals are idle. Whenever the program enters the SLEEP state, the MCU is configured to go into Power-down mode[18]. In this mode the main clock is turned off, however, Timer 2 is still enabled, thus keeping the RTC running. In this mode, the MCU draws significantly less power compared to when it is running. In this case, the device can only wake up when either an external interrupt or a Timer 2 interrupt is triggered. The way these are triggered is explained in section 5.3.6 and section 5.3.7 respectively.

IDLE state

Upon waking up, the MCU enters the IDLE state. In this state the MCU checks for 3 different things: is there an accelerometer-event? If so, change the state to DATA

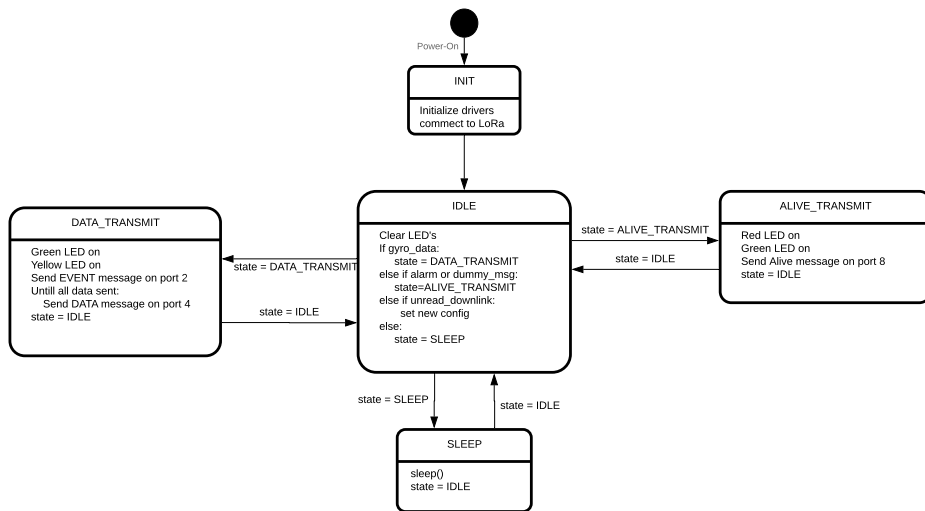


Figure 5.8: State diagram showing the state flow in during normal operation.

TRANSMIT. Is there an alarm or has the dummy button been pressed? If so, change the state to ALIVE TRANSMIT. Is there an unread downlink? Depending on the port number of the unread message. It can be a configuration message, to either set the sensitivity or motion interrupt threshold for the MPU6050. The configurations are set using either port 3 or port 5. Thus when receiving a downlink message, the port number of that message determines which configuration parameter is set. The threshold takes a value between 0-255 and the sensitivity between 0-3. It is worth noting that a device can only receive a downlink after sending an uplink. However if several downlinks are queued during the DATA TRANSMIT state, the last downlink is the only message that will be read. Every other downlink will be overwritten and consequently lost. The operation in IDLE mode is illustrated in fig. 5.9.

Alive Transmit state

Every 15 min the device will wake up to send a "still alive" message. This is to keep track of whether the device is still functioning as intended or not. During the ALIVE TRANSMIT state, the MCU enables the ADC and the MPU6050-temperature sensor to read one sample. After disabling both of them, the MCU creates an EVENT uplink message. This EVENT uplink message is sent as a confirmed LoRa uplink message(see section 3.2.3). The EVENT uplink message is sent on port 8 and contains a 32-bit UNIX timestamp, the

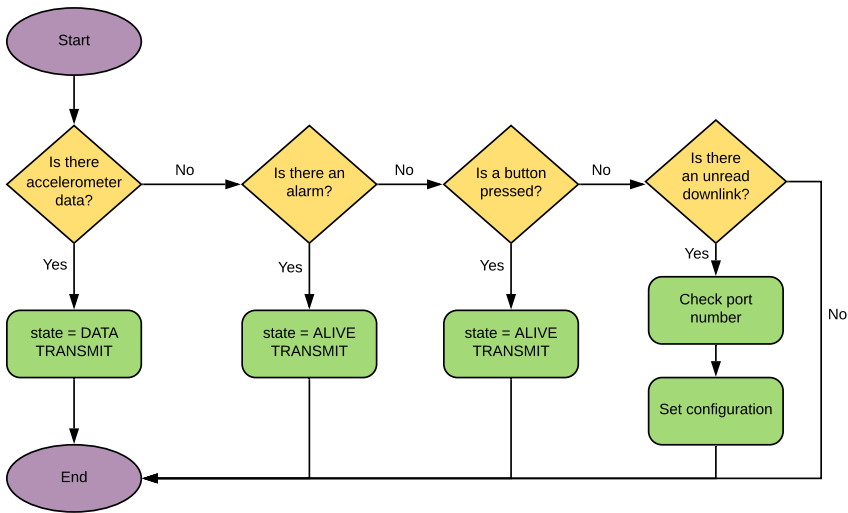


Figure 5.9: Flowchart illustrating the operations in IDLE state.

current device-battery level, the light level, temperature and lastly the raw accelerometer data from all three axes. The LoRa message payload is organized in the same way as the frame shown in fig. 5.10.

Event Uplink Payload 13 Bytes										
Timestamp 4 Bytes(32-bit unsigned integer)				Battery level 1 Byte	Light level 1 Byte	Temperature 1 Byte	Raw Acc data X-axis 2 byte	RAW Acc data Y-axis 2 byte	RAW Acc data Z-axis 2 byte	
bit 24-31	bit 15-23	bit 8-15	bit 0-7				bit 8-15	bit 0-7	bit 8-15	bit 0-7

Figure 5.10: An illustration of the ALIVE message frame.

"Raw" means that the data is not corrected with any offsets, its read as-is from the MPU6050-accelerometer. These readings can be used to determine the current positioning of the device(vertical, horizontal, up-side-down, etc.). The timestamp represents the exact local-time the message was created, this will vary from the server time depending on synchronization accuracy and clock skew. The message is broken down into bytes before the radio transmits it. A flowchart illustrating the full sequence in the ALIVE TRANSMIT state is provided in fig. 5.11.

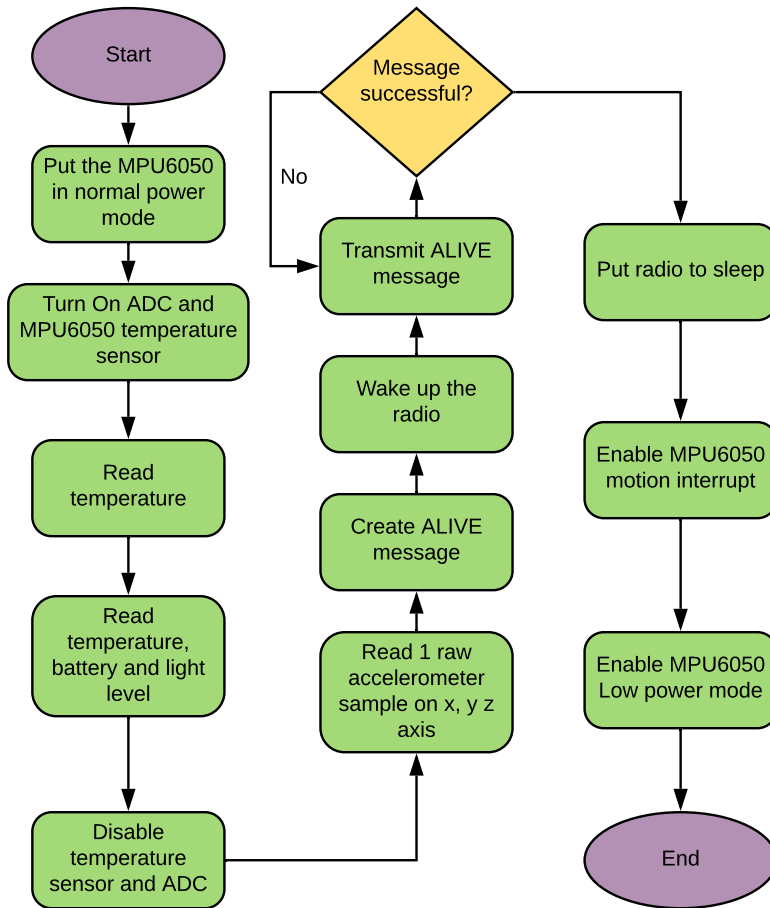


Figure 5.11: Flowchart illustrating the operations in ALIVE TRANSMIT state(FIFO refers to the embedded FIFO buffer on the MPU6050).

Data Transmit state

Whenever the MCU enters the TRANSMIT DATA state, this means that an event has occurred and there is data ready for transmission. This is done simply by sending an EVENT-message on port 2 followed by several APPEND DATA messages on port 4. The reason everything is not sent in one go is that the first the max payload in a LoRa-message is 255 B. Moreover the LoRaWAN Regional parameters[4] states that a message sent with an $SF = 12$ cannot be more than 51 B long. Thus the data has to be transmitted by sending

multiple messages.

The EVENT-message is built up in the format seen in fig. 5.12.

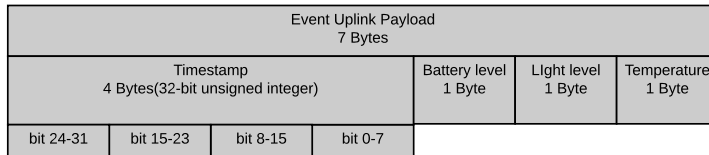


Figure 5.12: An illustration of the EVENT message frame.

The timestamp is as earlier mentioned in section 5.3.6 stored when the ISR is triggered. The battery level, light level and temperature is sampled upon the message being made.

The APPEND DATA-message consist solely of Accelerometer data, these data points are in comparison to the ALIVE-message corrected by the offset, thus giving us relative correct data measurements. To preserve the measurement resolution and to decrease processing overhead, these values are not converted. Each message contains 24 16 bit values, making the total payload length 24 B. In normal mode, the MPU6050 is operating at a 20Hz sampling rate, and the FIFO buffer can hold 1024 B of data. The first 8 s has a total size of 960 B. This means that 20 APPEND DATA messages containing 48 B has to be sent. Each message is sent as a confirmed uplink, this is to ensure that no data is lost. Each APPEND DATA message follows the format given in fig. 5.13.

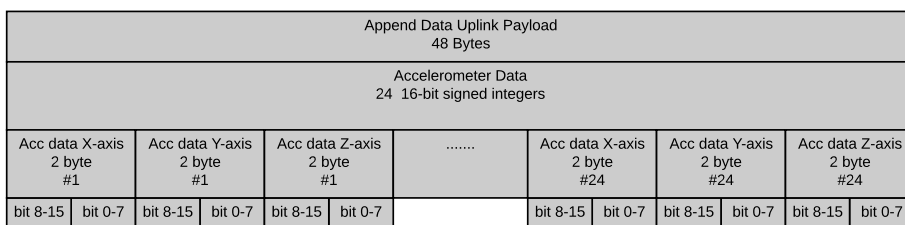


Figure 5.13: An illustration of the APPEND DATA message frame.

The complete scheme is illustrated in the flowchart given in fig. 5.14

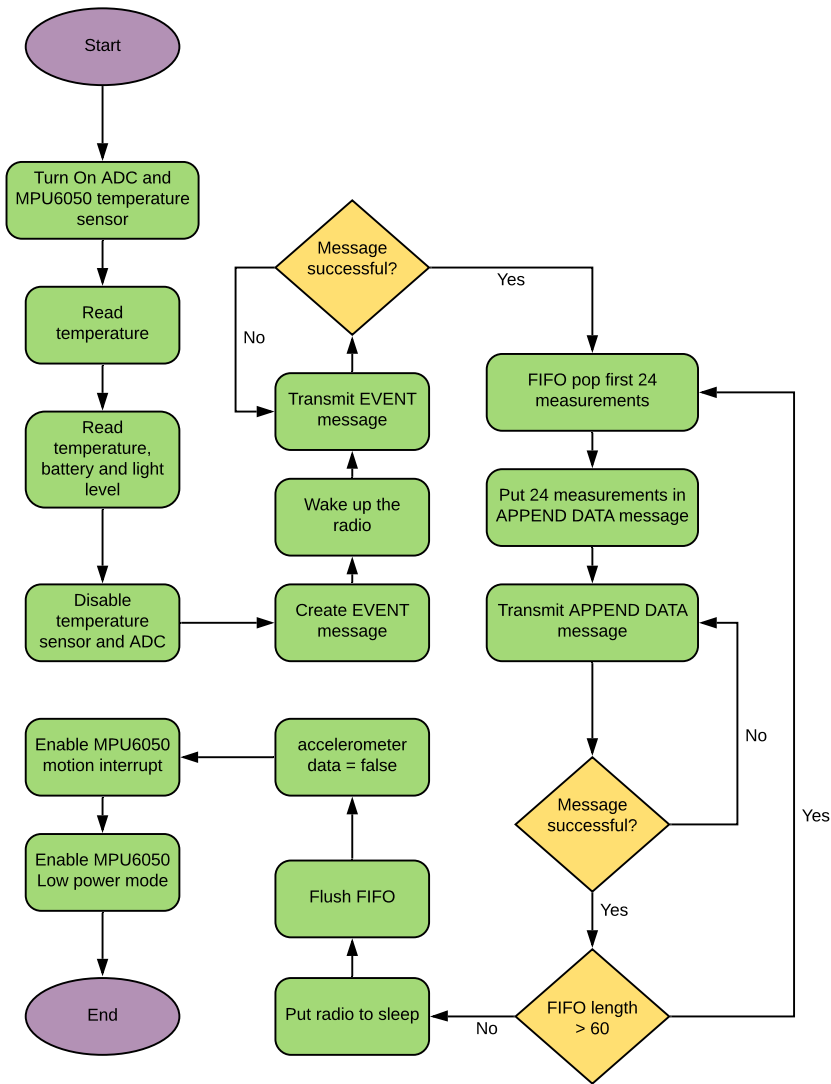


Figure 5.14: Flowchart illustrating the operations in DATA TRANSMIT state(FIFO refers to the embedded FIFO buffer on the MPU6050).

5.3.9 OTA Synchronization

The OTA(over the air) synchronization routine is not implemented in the main program, this is because of the earlier mentioned limitations of the public network(see section 4.5.2). However, the routine can easily be implemented as an own state in the main program. Upon

engaging in a sync-message, the MCU starts a local timer. This timer is to estimate the total time from the message being sent until it was received. The synchronization message is an ALIVE message sent on port 1 instead of port 8. This is sent as a confirmed uplink. Upon receiving the answer the timer is stopped and the server timestamp is extracted together with the callback-time(eq. (3.3)). The timestamp is then corrected by the calculating T_{skew} as in eq. (3.6). The downlink message has the frame-format described in fig. 5.16. This was implemented as a separate main-program, and is illustrated in fig. 5.15.

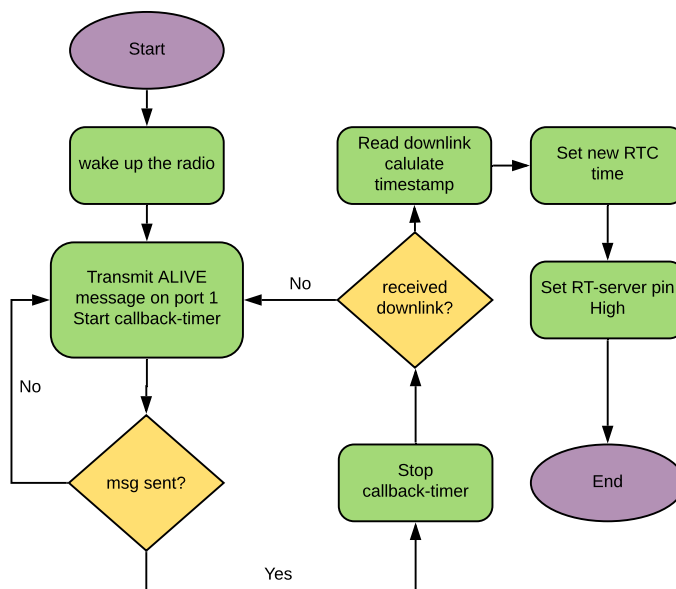


Figure 5.15: Flowchart illustrating OTA-synchronization on the end-device.

Sync Downlink Payload 10 Bytes									
Global Timestamp in us(T_{new}) 7 Bytes (56-bit integer)							Callback time in us(T_{loc}) 3 Bytes(24-bit integer)		
bit 48-55	bit 40-47	bit 32-39	bit 24-31	bit 16-23	bit 7-15	bit 0-7	bit 16-23	bit 7-15	bit 0-7

Figure 5.16: Frame of the downlink message received during OTA-synchronization.

5.4 Private gateway

The private gateway used was *Multitech's* MTCDTIP-LEU1. This is a robust gateway with support for 4G, 3G, IP67 and GPS[23]. The gateway is powered by PoE(power over ethernet) and can be configured using a web-GUI(graphic user interface).

5.4.1 LoRa network configuration

Through the GUI it was configured as a private gateway. The LoRa network can be customized in almost any way, however, in this case, it was configured following the EU863-870 MHz frequency-plan[4]. The only exception being the downlink queue. The downlink queue is configured to hold at a max 1 downlink message for any device. This means that if a callback misses it's downlink-window, no downlink is sent. A snapshot of the configuration is provided in Appendix C1. The gateway is also set to use the internal join server(see section 3.2.4), a snapshot is provided in Appendix C2. This turns the gateway into both a network-server and a join-server. Upon receiving an uplink it is now forwarded to the internal *Node-RED* application, which is continuously running on the gateway. *Node-RED* is a visual/graphical programming tool based on *Node.js*.

5.4.2 Gateway SW

The software on the gateway is made in Node-RED. The inner workings are quite simple: it receives and processes uplinks and downlinks. The program has three different processes: Uplink forward, downlink forward and RTT-response. These are further explained in the following sections.

Uplink forward

Upon receiving an uplink, the meta-data is formatted and a JSON-object is created. The JSON-object contains the following information:

This is then parsed to a string and sent using UDP to the RT-server. A snapshot of the *Node-RED* program can be seen in fig. 5.17

- **freq**: Which frequency the uplink was sent on.
- **datr**: The message data-rate/spreadfactor.
- **size**: The payload size in bytes.
- **port**: The uplink port.
- **deveui**: The EUI for the device which sent the uplink.
- **payload**: The message payload in bytes.
- **TpUL**: The gateway process time in ms(see section 3.2.5).

Table 5.1: Uplink metadata sent to the RT-server.



Figure 5.17: A snapshot of the code from the *Node-RED* application, showing the uplink-forward procedure.

Downlink forward

A potential callback-message from the RT-server is received in the form of a JSON object. The gateway parses this object upon reception and converts the message payload from string to bytes. The bytes are then aired as a downlink-message. A snapshot of the program is provided in fig. 5.18.



Figure 5.18: A snapshot of the code from the *Node-RED* application, showing the downlink-forward procedure.

RTT response

The RTT-response procedure is used to answer the RTT call from the RT-server. This is for the RT-server to calculate the RTT of a single UDP packet. Upon receiving a message the gateway responds by sending the exact same message on a different port to which the RT-server is listening. The only processing time required is upon reception and sending, this should give a good RTT-estimate. A snapshot of the code is provided in fig. 5.19.



Figure 5.19: A snapshot of the code from the *Node-RED* application, showing the RTT response procedure.

5.5 RT server

The RT-server was implemented using the NUCLEO-F411RE dev-board. The NUCLEO-F411RE from *STMicroelectronics* features an ARM STM32F411RE MCU and supports the RT-operating system *Mbed-OS* developed by *ARM*. The following sections covers the implementation of hardware and software respectively.

5.5.1 Hardware

The RT-server comprises of 2 main components: The NUCLEO-F411RE and an ENC28J60 Ethernet module. The NUCLEO-F411 is connected to a COM-port on a computer, that way it can print to a terminal using UART. The ENC28J60 is connected via an Ethernet cable to an Ethernet-switch, the GW(gateway) is connected to the same Ethernet-switch. The switch is necessary because it supports PoE which the GW requires. The complete setup can be seen as illustrated in fig. 5.20.

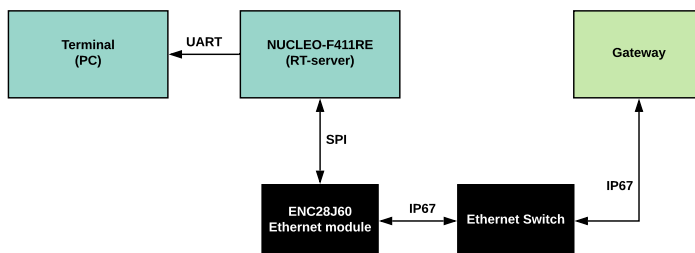


Figure 5.20: Collaboration diagram showing the RT-server and it's peripherals.

5.5.2 Software

The entire software is written in C++ using Mbed-OS, *github*² was used to document development. All the drivers except the ENC28J60-driver are already present in the OS. The ENC28J60-driver is imported from the library: *UIPEthernet*[24]. This is available from the Mbed online community. The UIPEthernet-library provides support for TCP, UDP, IP4, IP6 and more. This is exactly what is needed in the RT-server. IT was decided to use UDP to communicate between the server and GW. Since UDP is known to be faster than TCP, choosing UDP will reduce the total RTT. In addition to better RTT, it will increase the chance of a callback catching the downlink-window.

To synchronize the RT-server with the local time, it was decided to use NTP(Network time protocol) which is described in RFC 958. Upon booting, the server will send an NTP request via UDP on port 123. The corresponding answer will contain a 32-bit UNIX-timestamp. The scope of the task is to investigate synchronization between the RT-server and the end-device, thus the clock skew between the RT-server and time server is irrelevant. In our system the RT-server is the master-clock, therefore if the server deviates from the time-server, this won't affect our investigation.

Mbed-OS provides a driver for the STM32's RTC. However, the RTC only works with seconds, to be able to truly investigate clock drift and deviation it was decided to use one of the 64-bit timers as an RTC instead. The 64-bit timer works with μs resolution. Upon receiving the time from the time server, the timer is started and the received timestamp stored. Thus to read the current time, the value of the timer is added to the stored timestamp, resulting in the current time. This results in a 64-bit UNIX timestamp in μs resolution.

Time-server synchronization

Before initiating the main program, the RT-server synchronizes with a timeserver. This is done by posting an NTP request on port 123 of the timeserver. The server then proceeds to wait for an answer. If no answer is received after 3 seconds, the NTP-call is re-sent. The sequence is illustrated in fig. 5.21.

²Repository available at https://github.com/tobulf/Master2019_RT-server

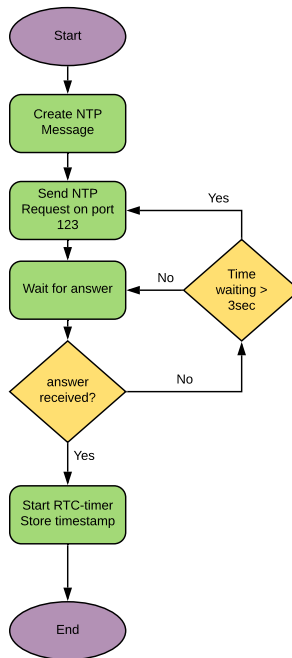


Figure 5.21: Flowchart illustrating the time-server synchronization sequence.

Main program

The RT-server enters the main program immediately after synchronizing with the time server. There are no defined states in this program, it only responds to synchronization messages. Upon receiving an uplink, the RT-server checks if it is a sync message. If it is, run the sync procedure, if not: do nothing. The callback sequence is illustrated in fig. 5.22. The uplink-airtime is calculated using eq. (3.12). The gateway process time is a part of the message metadata(see table 5.1). The RTT-timer and Callback-timer are used to estimate the RTT and process time T_{GC} in eq. (3.3). The total time spent so far (T_{tot*}) is then calculated using eq. (3.3). The timestamp and T_{tot*} is then parsed to JSON and sent back to the gateway as a callback. The gateway will forward the callback and send it as a downlink to the end-device.

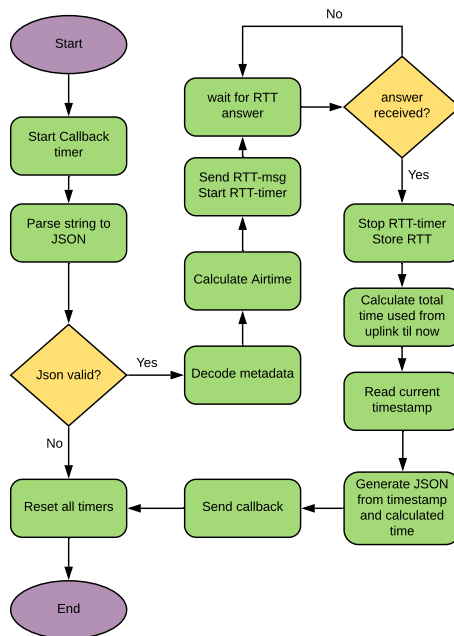


Figure 5.22: Flowchart illustrating the callback-sequence upon receiving a sync-message from an end-device.

5.6 TTN back-end

The TTN back-end was set up using the TTN-console and TTN’s python SDK(Software development kit)[25]. *Github*³ was used to document the code. The console was used to set up an application server, to which several end-devices can connect. The console works as an MQTT-broker, such that a corresponding client was implemented as well. The client receives data from the end device and writes these to file. These files are to be used for data analysis at a later stage.

5.6.1 TTN console and SDK

The TTN-console is a GUI for configuring application-servers. A snapshot of the GUI is provided in Appendix A1. This is where all the devices are added to the join-server. This is very simple and straight forward since TTN generates all EUI’s and keys automatically.

³Repository available at https://github.com/tobulf/Master2019_TTN-server

The application-server also works as an MQTT broker and the corresponding client can be implemented using libraries provided by TTN[25]. In this case, the MQTT-broker was implemented in *Python*. Using TTN’s Python SDK(Software development kit)[25] to make a simple MQTT broker is straight forward. The client can then subscribe to the application-server and receive uplinks from all end-devices attached to said application. Furthermore, the application server enables the user to send downlinks directly to an end-device, this was used to configure the end-devices after deployment. This can be seen in Appendix A2.

5.6.2 MQTT-client

The data collection is done within the callback function in an MQTT-client. This client subscribes to our application-server at TTN. Thus every uplink sent to the application server will be forwarded to the subscribing MQTT-client. This client is running on a personal computer at the office at NTNU. Each time the client receives an uplink the data from the end-devices are sorted depending on the port number and written to file. For each day that passes, a new file will be created. This keeps the files from growing large. Each message is timestamped using the servers clock. The files are in *.csv* format where each value is comma-separated. Each line in the file represents either an event or an alive message. Depending on the message type(event or alive) each line follow either the format in fig. 5.23 or fig. 5.24.

Alive message								
msg port number	Server timestamp (Unix)	End-device timestamp (Unix)	Battery level (in percent)	Light-level (in percent)	Temperature (in celsius)	Raw Acc data X-axis	Raw Acc data Y-axis	Raw Acc data Z-axis

Figure 5.23: Frame showing the comma-separated ALIVE-message values in the log-files. The port number here is always 8.

Event data												
msg port number	Server timestamp (Unix)	End-device timestamp (Unix)	Battery level (in percent)	Light-level (in percent)	Temperature (in celsius)	Acc data #1 X-axis	Acc data #1 Y-axis	Acc data #1 Z-axis	Acc data #160 X-axis	Acc data #160 Y-axis	Acc data #160 Z-axis

Figure 5.24: Frame showing the comma-separated data of an event. The port number here is always 2.

No end-device will be synchronized with the MQTT-client, thus the end-device timestamp

will represent the total time the device has been active. This is because of the limitations mentioned in section 1.3 and section 4.5.2.

Testing and results

This section covers the testing and results of the system. The embedded-system was tested and verified against the acceptance criteria in section 4.4. In addition to plotting the collected vibration-data, two other tests have been done: OTA-synchronization and battery lifetime-expectancy. Each of these will be covered by their respective section. Furthermore, tests have been conducted to verify the calibration scheme, server-synchronization, end-device clock-drift, and end-device configuration.

6.1 Test procedure

6.1.1 Hardware test

An oscilloscope was used to test both voltage levels, UART and I2C. Furthermore, the UART was tested by connecting it to a COM-port and writing to a terminal. This was also used as a debug-tool. The ADC was tested by using the light-sensor and an oscilloscope. The battery-level circuit was verified using a voltage supply on varying voltages. The LoRa capabilities were verified by sending messages to TTN-console[10]. One fault was found and corrected. It was the I2C lines between the MCU and MPU6050. The internal pull-ups on the Atmega did not work as the internal pull-up resistance was to great. This was solved by making jumper with 2 1.5 k Ω resistors from the header J11 to VCC on J1.

6.1.2 OTA synchronization

In order to test the accuracy of the synchronization scheme, following setup was applied:

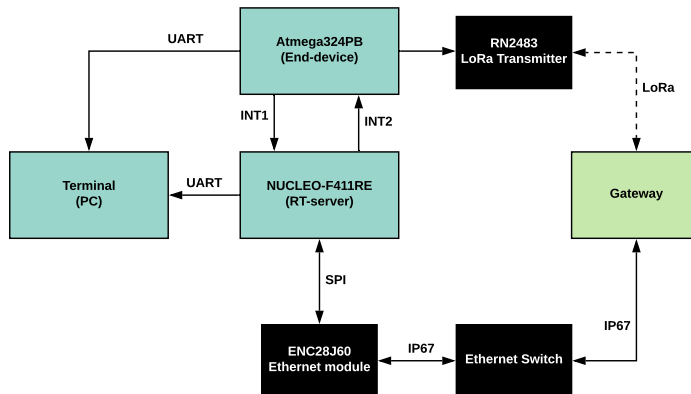


Figure 6.1: Collaboration diagram showing the OTA-synchronization test-setup.

In this scheme, only one end-device is used, together with the RT-server and the private gateway. Upon synchronization, the End-device set a pin to high(See INT1 in fig. 6.1), this is connected to the RT-server. Upon sensing the INT1, the RT-server stores the current timestamp and toggle a different pin(See INT2 in fig. 6.1), this pin causes an interrupt on the end-device on rising or falling edge, the RT-server then proceed to print the timestamp for logging. Sensing the incoming interrupt causes the end-device to store the current timestamp and print it for logging. After printing the timestamp the MCU resets the INT1, letting the RT-server know it has printed the timestamp. This is to ensure that neither the RT-server nor the end-device writes to log unless both are ready. This way the 2 different logs will be symmetric. The logs are written using a terminal with capture-abilities. The devices uses *printf* to write a .csv formatted string on each sync. The ISR is illustrated in fig. 6.2. Upon measuring the mean, an attempt on improving the synchronization was made. Simply by subtracting/adding the mean-difference to the applied synchronization value. The scheme was tested using two different Data-rates(DR): DR-0 and DR-5(see table 3.3). The data was plotted using *Matlab* and the results can be seen in section 6.2.1.

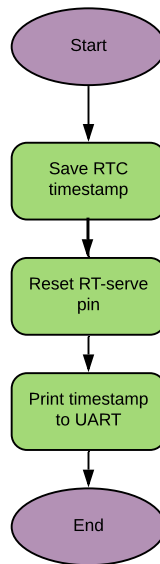


Figure 6.2: Flowchart illustrating end-device ISR for printing the timestamp.

6.1.3 Battery lifetime expectancy

The power consumption has to be estimated to calculate the expected battery life of the end-device. This was done with a *Otii* from *QOITECH* with a 4000 Hz sampling rate. To check the power consumption, the battery was removed and the *Otii* used as a power source. The power was measured in different states:

- Average power consumption during ALIVE TRANSMIT state
- Average power consumption during DATA TRANSMIT state
- Average power consumption during SLEEP state
- Average power consumption during the IDLE state

In addition to this, the power-consumption during sleep state, active state and transmit state for $DR = 0$ and $DR = 5$ has been measured. In this configuration the end-device was configured to send a 10 B unconfirmed message and the accelerometer sample-rate reduced to 1.25 Hz. In this configuration, the device can be regarded as a simple end-device and uses as little energy as possible. All the results can be found in section 6.2.2.

6.1.4 Vibration data-collection

To collect vibration-data, three devices was deployed on a office-door. They were deployed in the following way:

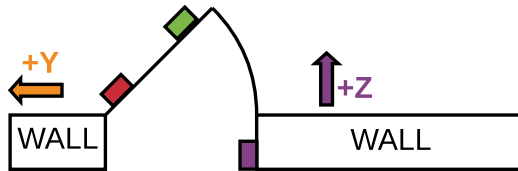


Figure 6.3: Illustration of how the devices were mounted on the door, device 1(Red) and device 2(Green) was mounted directly on the door, while device-3(Purple) was mounted in the door-sill.

The devices were all mounted inline with the wall and with the accelerometer X-axis pointing to the floor. The different axis are illustrated in fig. 6.4.

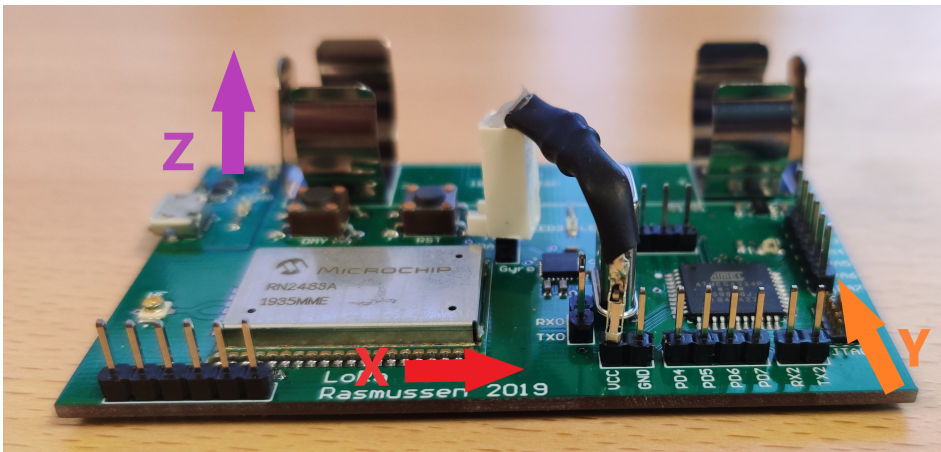


Figure 6.4: Illustration of the X(Red), Y(Orange) and Z(Purple) axis positive direction in comparison to the end-device.

Devices 1 and 2 will be exposed to a decent amount of force compared to device 3. Device 3 was mounted in the door-sill, which is a different surface. The door sill is also stiffer, thus smaller more high-frequent forces. Therefore device 1 has a sensitivity of $\pm 4G$ and 40 Hz sample-rate, device 2 has a sensitivity of $\pm 8G$ and 40 Hz sample-rate and lastly,

device 3 has a sensitivity of $\pm 2G$ and 166 Hz sample-rate. This means that device 3 records approximately 2 seconds of data. The data was collected using the TTN-server and MQTT-client on a local computer. The data was written to *.csv* files. Plots have been made in *Matlab* using the received data. The results can be found in section 6.2.3.

6.1.5 End-device clock drift

The clock drift was tested by synchronizing two end-devices by using an interrupt pin on interfaced with the RT-server. The RT-server set the pin high on a pre-determined time to synchronize the devices. The synchronized end-devices were then put away for 1 month. After this time the RT-server and the end-device printed their clock-time at the same time. The result can be seen in section 6.2.6.

6.1.6 End-device remote configuration

The configuration of the end device was verified using the TTN application-server GUI. An end-device was configured by en-queueing downlinks on port-5, this configures the gyro-sensitivity. The result can be seen in section 6.2.7.

6.2 Results

It's worth mentioning that not all devices have been deployed during the tests. However, all devices were tested in terms of hardware. Out of 15 devices, one was found to be faulty. The data-collection was verified since the MQTT-client stored the payload of EVENT and ALIVE messages to *.csv* files, each data-entry was timestamped with a server-time and event-time. The end-devices was triggered whenever the office door was opened or closed. The simulate-downlink in the TTN-console was used to set new wake-up thresholds on the deployed devices.

```
8,1580894382,316251,92,2,16,-7891,385,-727
8,1580895282,317152,92,4,16,-7873,357,-743
```

Table 6.1: Alive messages from device 1(section 6.1.4) stored by the MQTT-client. The above example follows the format in fig. 5.23.

2,1580895667,317528,92,4,16,33,-106,26,16,-117,27,20,-115,16,15,-123,-30,...
 2,1580895811,317672,92,4,16,-54,-44,69,70,-114,-283,-87,-206,-26,215,-109,...

Table 6.2: Vibration data from device 1(section 6.1.4) stored by the MQTT-client. This is just some of the data from 2 events and follows the format in fig. 5.24.

6.2.1 OTA synchronization

This section covers the results of the OTA-synchronization. 100 synchronization messages was sent using $DR = 5$ and $DR = 0$ (fig. 6.5 and fig. 6.9). After 100-messages, the mean was calculated to see if there was a constant deviation(fig. 6.6 and fig. 6.10). After correcting for the constant deviation, another 100 messages was sent(fig. 6.7 and fig. 6.11). The clock-drift as seen from the end-device was plotted, and the accuracy of the synchronization calculated(fig. 6.8 and fig. 6.12). For $DR = 5$ the error was estimated to ± 223 ms, this can be seen as the yellow red and green lines in fig. 6.8. For $DR = 0$ the same error was estimated to ± 140 ms, this can be seen in fig. 6.12.

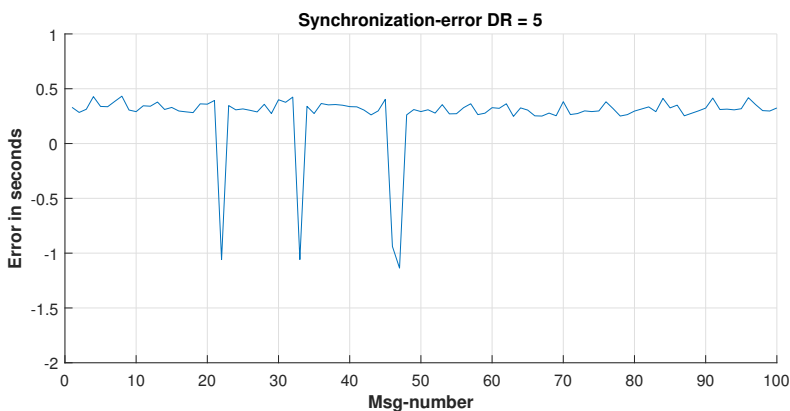


Figure 6.5: Time series showing the skew of the end-device compared to the server during 100 synchronizations.

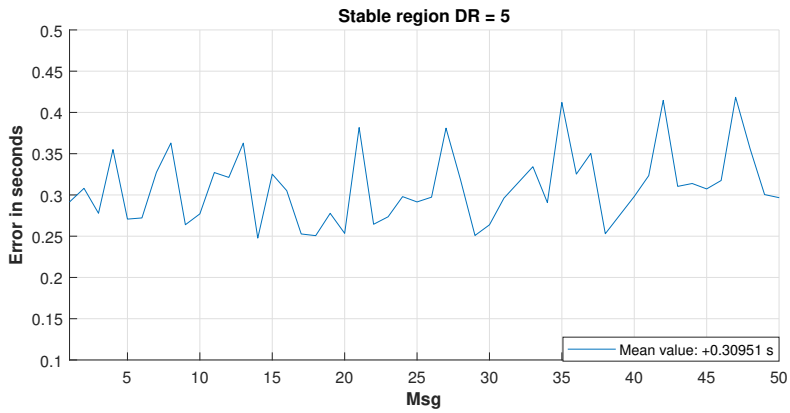


Figure 6.6: Time series showing the skew compared to server in the stable region(message 50-100 in fig. 6.5).

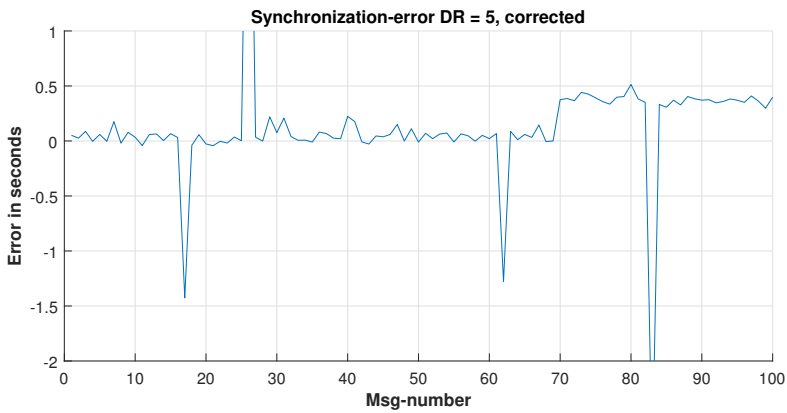


Figure 6.7: Time series showing the skew after compensating for the constant deviation.

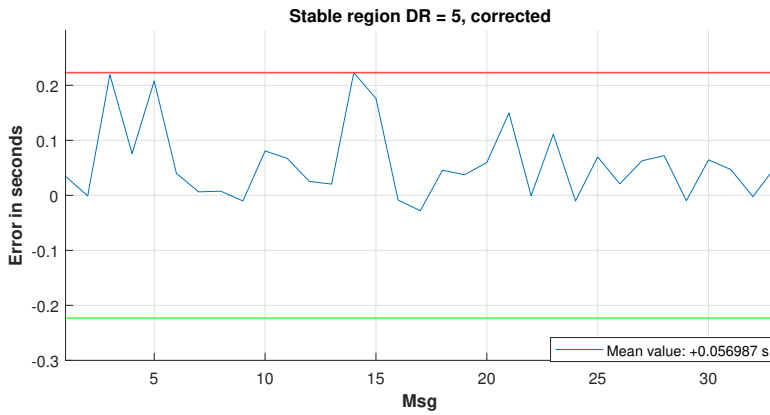


Figure 6.8: Time series showing the skew in the stable region(message 27-61 in fig. 6.7).

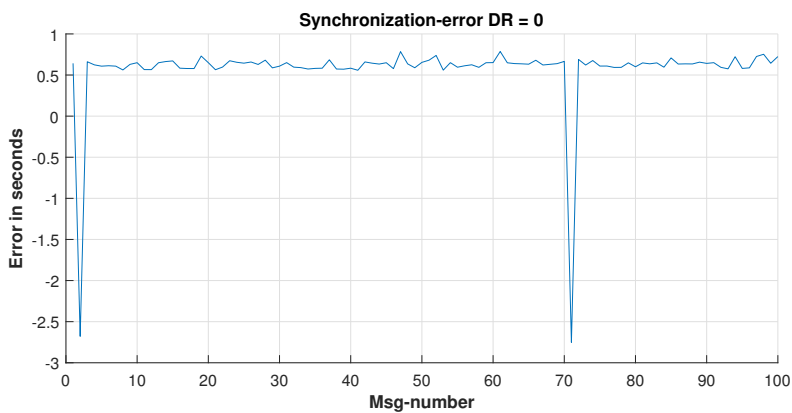


Figure 6.9: Time series showing the skew of the end-device compared to the server during 100 synchronizations.

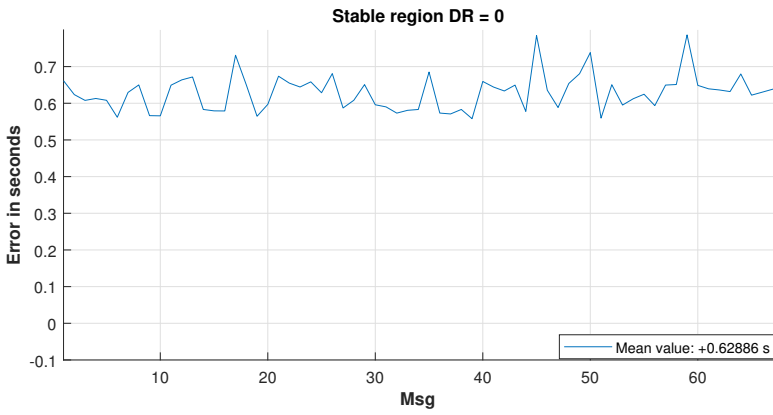


Figure 6.10: Time series showing the skew compared to server in the stable region(message 3-69 in fig. 6.9)

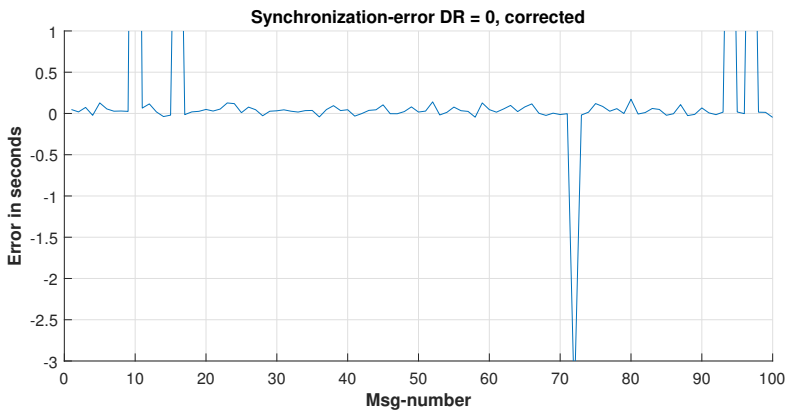


Figure 6.11: Time series showing the skew after compensating for the constant deviation.

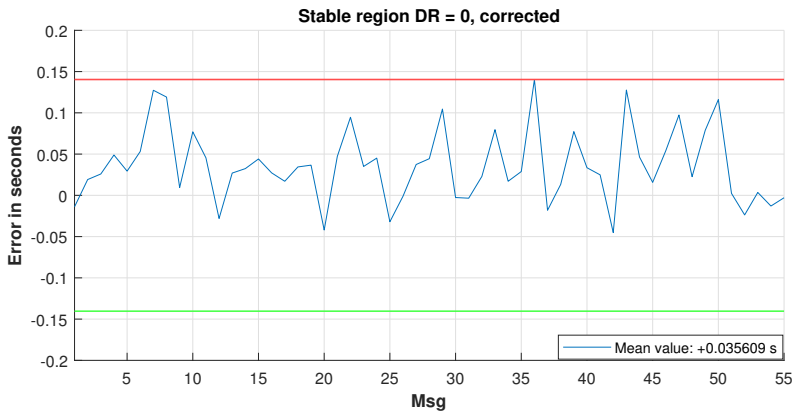


Figure 6.12: Time series showing the skew in the stable region(message 17-71 in fig. 6.11).

6.2.2 Battery lifetime expectancy

The battery lifetime expectancy can be calculated knowing the power-consumption of the end-device. The following sections presents the power consumption-data collected from the *Oiii*.

Power-Consumption while idle

power consumption-data collected throughout 16 h was used to estimate the average power-consumption while the end-device is idle. During this time the end device woke up every 15 min to send an alive message, before going back to sleep. The measured average power consumption was 651 μA . fig. 6.13 shows the consumption during one such message.

Power-Consumption while sending vibration data

Power-data collected from 12 events has been analyzed to estimate the average consumption while sending the vibration data. This is the consumption during the TRANSMIT DATA state. The calculated average power consumption was 19.75 mA and the average sequence duration was 188.8 s. fig. 6.14 shows the consumption during one such sequence. During this sequence, the end-device sends at least 21 messages. The average battery consumption can also be written as 1.035 mA h.

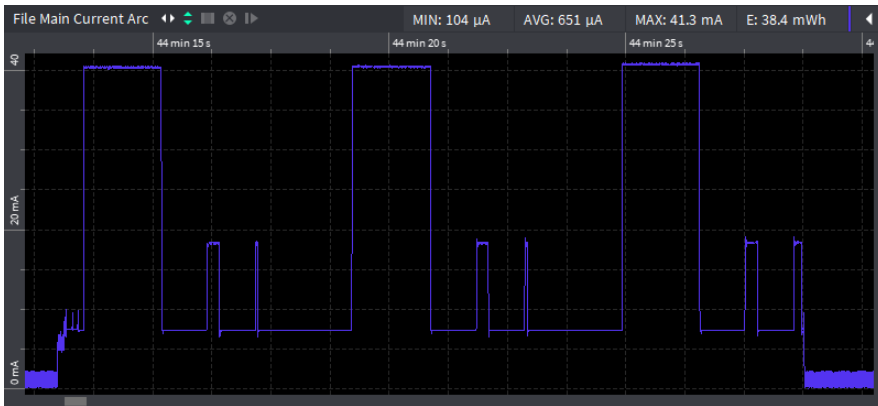


Figure 6.13: Time series showing the power-consumption during a single alive message.

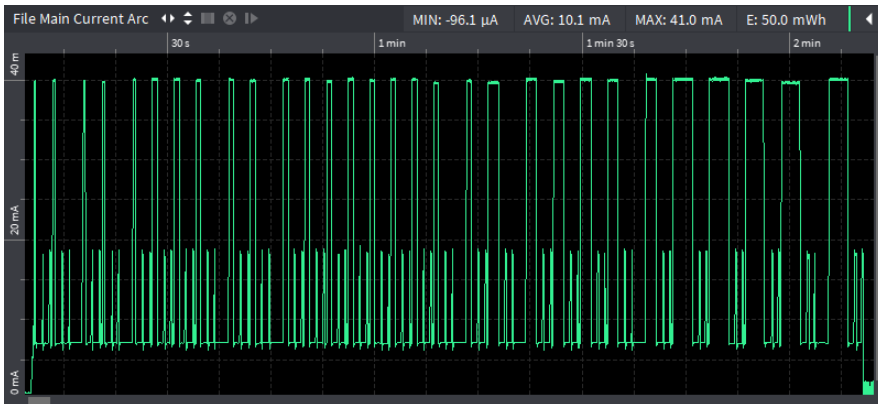


Figure 6.14: Time series showing the power-consumption while sending vibration data.

Power-Consumption during sleep

With the sampling rate of the MPU6050 set to 40 Hz. It consumed an average of 509 μA . When reduced to 1.25 Hz, the average became 326 μA . However during normal operation, the end device will have a 40 Hz sampling rate. A one second sample is provided in fig. 6.15. Furthermore fig. 6.17 shows the MCU waking from sleep mode, and fig. 6.18 shows the MCU going back to sleep mode.

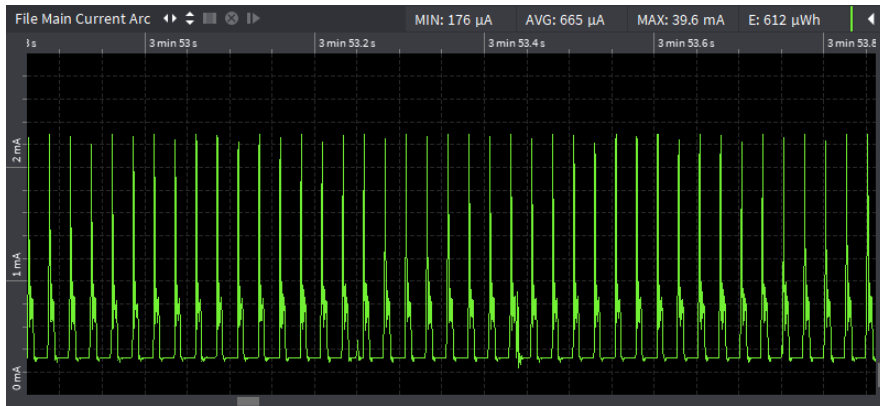


Figure 6.15: Time series showing power-consumption during 1 second. The device is in sleep-mode with 40 Hz sampling-rate on the MPU6050

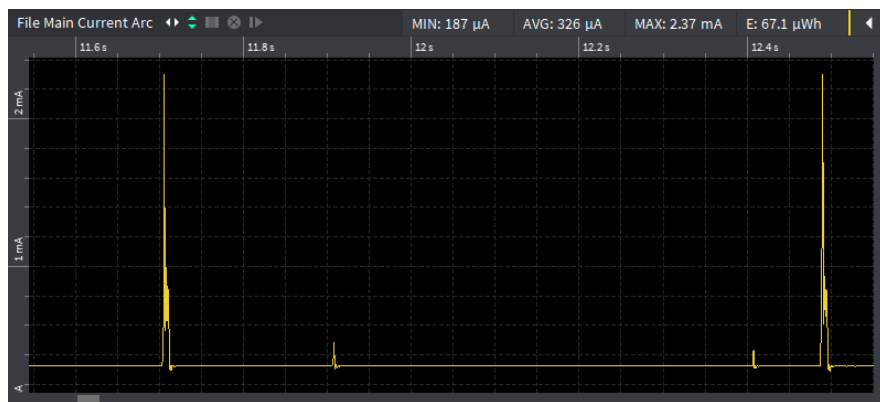


Figure 6.16: Time series showing power-consumption during 1 second. The device is in sleep-mode with 1.25 Hz sampling-rate on the MPU6050

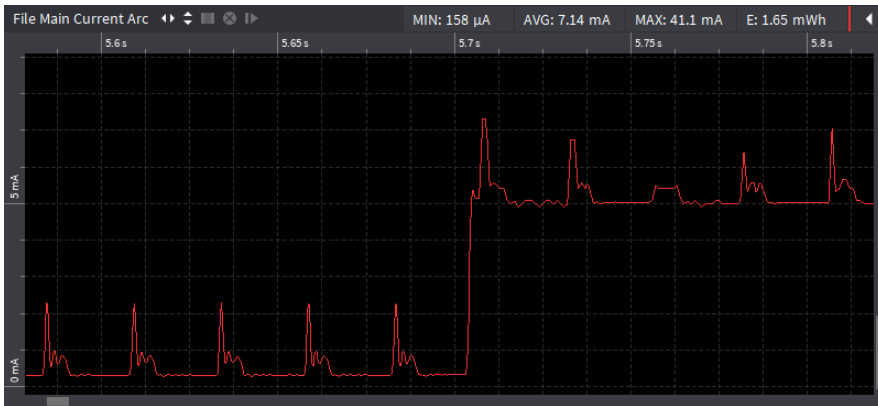


Figure 6.17: Time series showing power-consumption when the MCU wakes up from sleep mode. The device was in sleep-mode with 40 Hz sampling-rate on the MPU6050

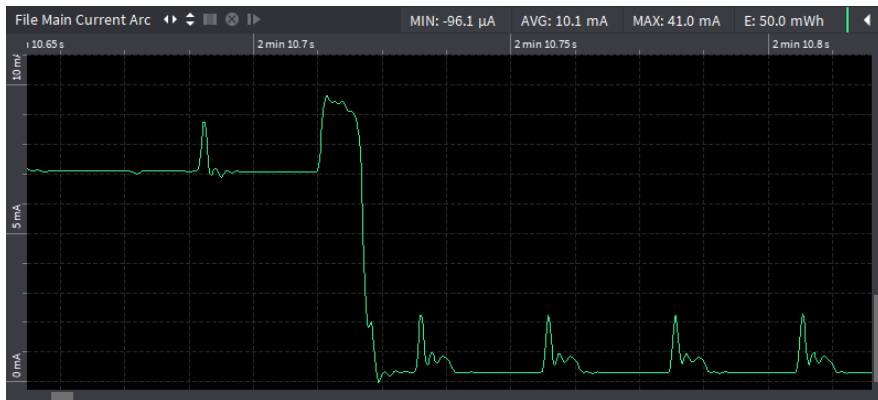


Figure 6.18: Time series showing power-consumption when the MCU is going back to sleep mode. The device goes into sleep-mode with 40 Hz sampling-rate on the MPU6050

As a simple LoRa end-device

The consumption was measured for $DR = 0$ and $DR = 5$. During transmission the average consumption was measured to 20.1 mA for a duration of 3.85 s for $DR = 0$ and 7.64 mA for a duration of 2.47 s for $DR = 5$. fig. 6.19 shows the consumption during one message on each of these Data-rates.

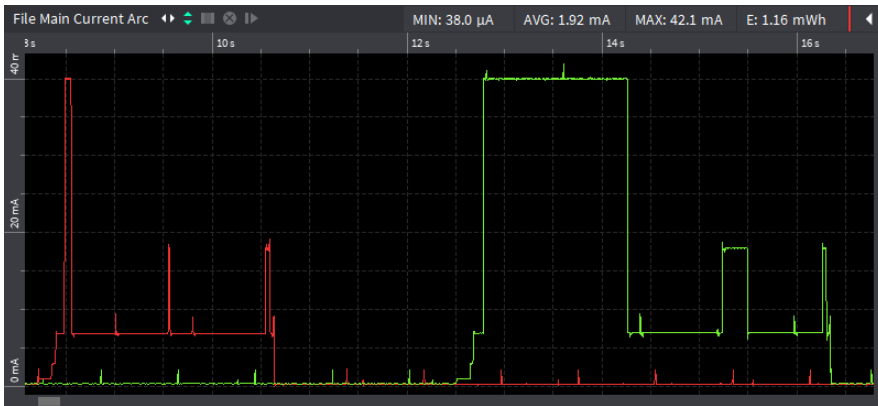


Figure 6.19: Time series showing power-consumption during 1 transmission with $DR = 0$ (green) and $DR = 5$ (Red).

6.2.3 Vibration data

The three devices(device 1, 2 and 3) were deployed for a duration of 15 days, during this time a lot of data was collected. They were deployed according to the setup in section 6.1.4. A sample of data from February 1st have been plotted in fig. 6.20, fig. 6.21, fig. 6.22 and fig. 6.23. This sample is an amalgamation of data from all three devices, using the server-timestamp for alignment. A plot of the light levels recorded during a period of 24 h has been provided in fig. 6.24.

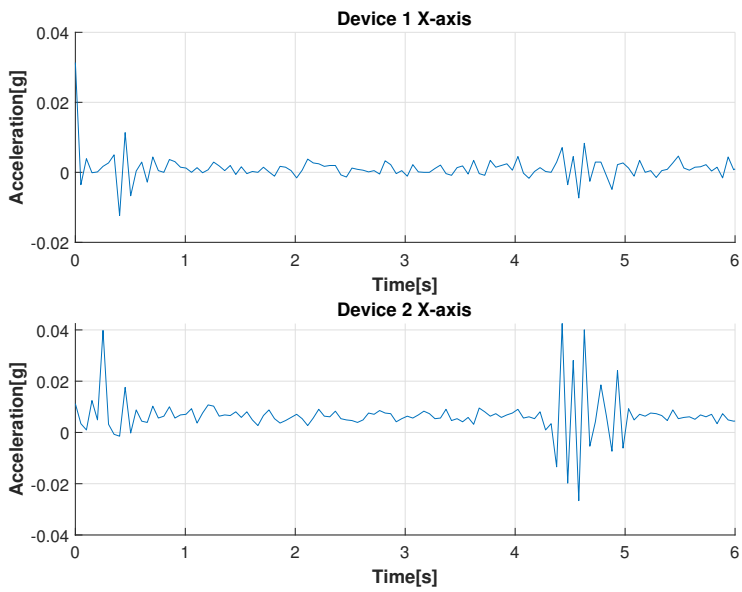


Figure 6.20: Time series showing the acceleration on the x-axis of device 1 and 2.

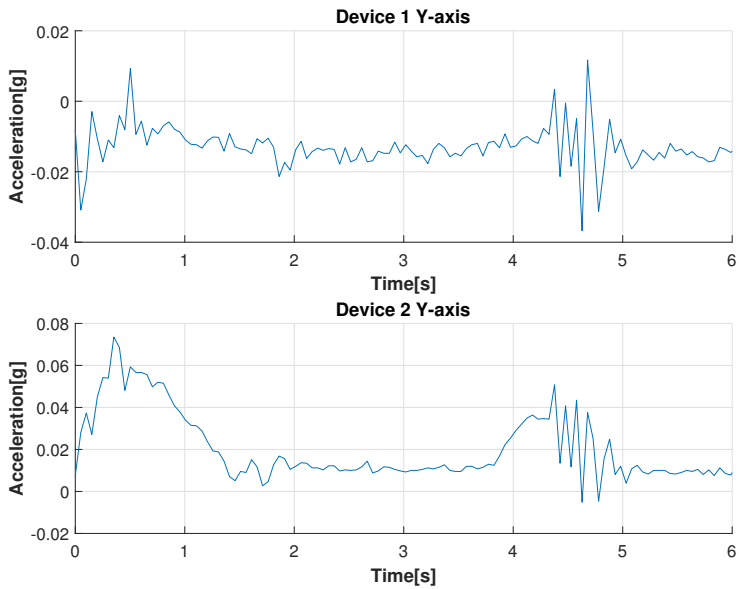


Figure 6.21: Time series showing the acceleration on the y-axis of device 1 and 2.

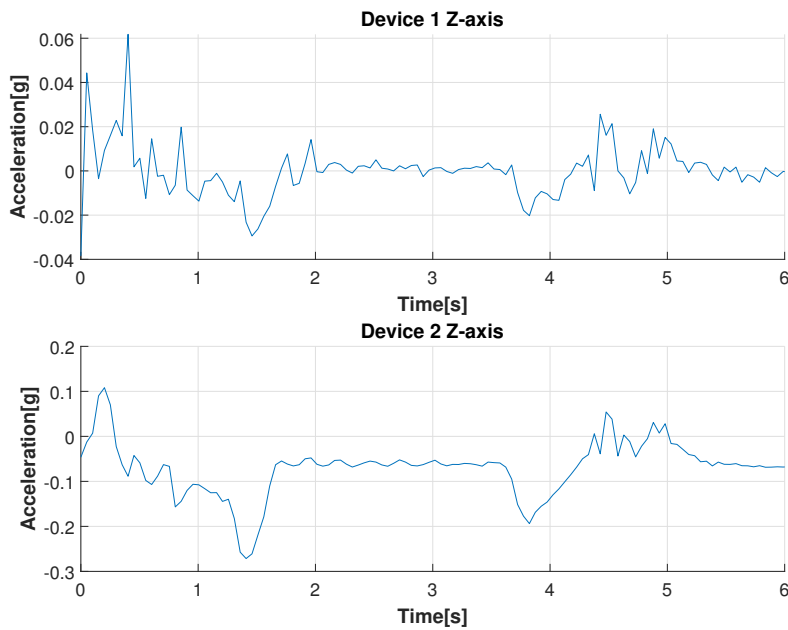


Figure 6.22: Time series showing the acceleration on the z-axis of device 1 and 2.

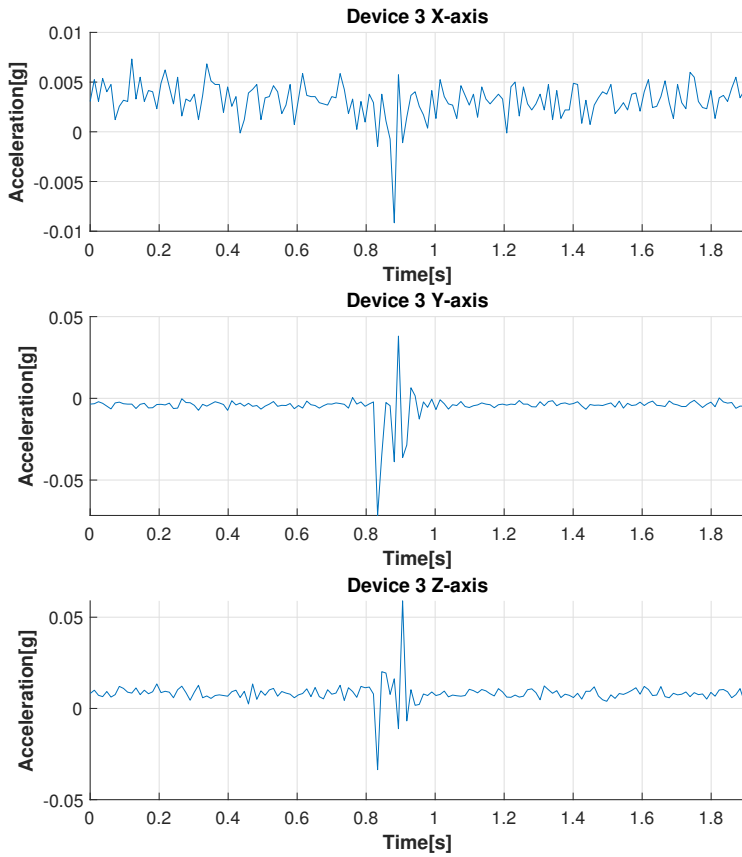


Figure 6.23: Time series showing the acceleration on all axis for device 3.

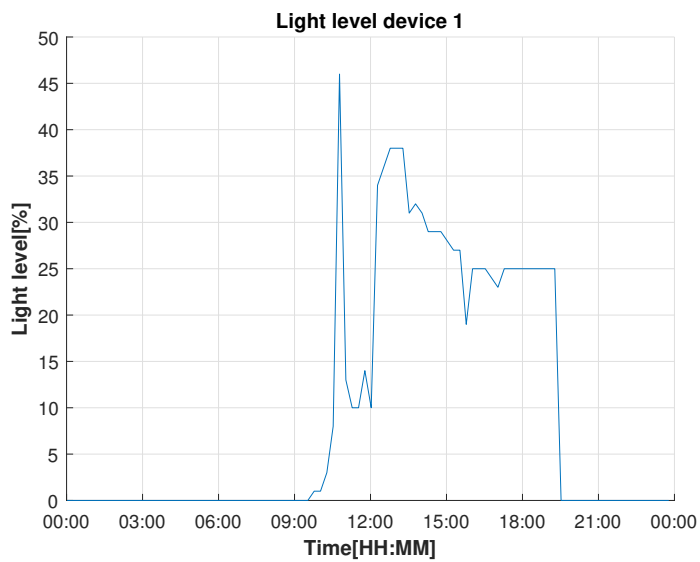


Figure 6.24: Time series showing the light level during 1 day for device 1.

6.2.4 Calibration results

The calibration scheme was tested by printing the accelerometer values to a terminal before and after the calibration. The results can be seen in the screenshots provided in fig. 6.25.

```

X: 9 Y: 20 Z: 4 Temp: 20 CRLF X: 384 Y: -1176 Z: 131 Temp: 32 CRLF
X: 27 Y: -38 Z: 28 Temp: 20 CRLF X: 394 Y: -1122 Z: 77 Temp: 32 CRLF
X: -13 Y: 2 Z: -18 Temp: 20 CRLF X: 374 Y: -1134 Z: 101 Temp: 32 CRLF
X: 7 Y: -12 Z: -40 Temp: 20 CRLF X: 418 Y: -1114 Z: 17 Temp: 32 CRLF
X: -1 Y: 4 Z: -26 Temp: 20 CRLF X: 436 Y: -1202 Z: 139 Temp: 32 CRLF
X: -21 Y: -20 Z: 36 Temp: 20 CRLF X: 336 Y: -1154 Z: 169 Temp: 32 CRLF
X: 7 Y: -4 Z: 42 Temp: 20 CRLF X: 438 Y: -1202 Z: 9 Temp: 32 CRLF
X: 1 Y: -6 Z: -30 Temp: 20 CRLF X: 378 Y: -1064 Z: 87 Temp: 32 CRLF
X: -13 Y: -12 Z: 0 Temp: 20 CRLF X: 430 Y: -1116 Z: 65 Temp: 32 CRLF
X: 61 Y: -34 Z: -40 Temp: 20 CRLF X: 434 Y: -1178 Z: 113 Temp: 32 CRLF
X: -7 Y: 18 Z: 32 Temp: 20 CRLF X: 416 Y: -1116 Z: 113 Temp: 32 CRLF
X: 29 Y: -6 Z: -34 Temp: 20 CRLF X: 372 Y: -1186 Z: 65 Temp: 32 CRLF
X: -13 Y: -22 Z: -28 Temp: 20 CRLF X: 360 Y: -1114 Z: 135 Temp: 32 CRLF
X: -25 Y: -14 Z: -46 Temp: 20 CRLF X: 420 Y: -1116 Z: 77 Temp: 32 CRLF
X: 5 Y: 18 Z: -4 Temp: 20 CRLF X: 416 Y: -1150 Z: 117 Temp: 32 CRLF

```

Figure 6.25: Snapshot of the terminal printing the accelerometer values before(right) and after(left) calibration.

6.2.5 Time server synchronization

The RT-server was able to synchronize by sending an NTP-request to a time-server. The snapshot in fig. 6.26 shows the debug data printed upon booting, the NTP synchronization can be seen at the beginning.

```

Initializing ethernet...CRLF
sending NTP packet...CRLF
Time synched, Epoch: 1577557351 CRLF
IP address: 129.241.187.18 CRLF
Netmask: 255.255.255.0 CRLF
Gateway: 129.241.187.1 CRLF

Listening on UDP, port 23073 CRLF
-----
Uplink received from 00-04-a3-0b-00-eb-9f-66 CRLF
Time GMT0: 18:23:03 PM CRLF

```

Figure 6.26: Snapshot of the terminal where the RT-server boot sequence is printed.

6.2.6 End-device clock drift

The timestamps upon synchronization and at the end of the test were printed to a terminal. This can be seen in the snapshots fig. 6.27 and fig. 6.28. Of the 2 devices only one survived for 1 month, the other device had a dead battery. The clock drift can be calculated using the timestamps provided in the snapshots.

```
Bat 97 % Time: 1576945463 s 801139 us ^ Seconds since January 1, 1970 = 1576945463, us: 254725
```

Figure 6.27: Snapshot of the terminal at the start of the clock-drift test(RT-server-time to the right).

```
Bat 61 % Time: 1579634200 s 357834 us ^ Seconds since January 1, 1970 = 1579632806, us: 934384
```

Figure 6.28: Snapshot of the terminal at the end of the clock-drift test(RT-server-time to the right).

6.2.7 End-device remote configuration

Upon receiving an configuration the end-device printed the new configuration parameter to terminal, this is shown in fig. 6.29. The en-queued downlink messages can be seen in fig. 6.30.

```
Config received on port 5  
Sensitivity set to 4G  
Config received on port 5  
Sensitivity set to 16G
```

Figure 6.29: Screenshot of the terminal when configuring the sensitivity of a end-device.

▲ 19:10:42	4	8	<i>retry confirmed</i>	payload: 00 00 00 A1 64 37 03 05 D8 02 E1 00 C1
▼ 19:10:37		0		
▲ 19:10:36	4	8	<i>confirmed</i>	payload: 00 00 00 A1 64 37 03 05 D8 02 E1 00 C1
▼ 19:10:33		5		payload: 03
▲ 19:10:31	3	8	<i>confirmed</i>	payload: 00 00 00 9C 64 22 03 05 F8 02 B8 00 BF
▼ 19:10:26		5	<i>scheduled</i>	payload: 03
▼ 19:08:52		0		
▲ 19:08:50	2	8	<i>retry confirmed</i>	payload: 00 00 00 32 64 38 14 0C 1D 04 B8 01 82
▼ 19:08:46		0		
▲ 19:08:45	2	8	<i>confirmed</i>	payload: 00 00 00 32 64 38 14 0C 1D 04 B8 01 82
▼ 19:08:42		5		payload: 01
▲ 19:08:40	1	8	<i>confirmed</i>	payload: 00 00 00 2D 64 22 14 0B D3 05 2C 01 0E
▼ 19:08:35		5	<i>scheduled</i>	payload: 01

Figure 6.30: Snapshot application server GUI showing the en-queued configuration downlinks on port 5(Row 4).

Discussion

This chapter presents a discussion covering the system results in accordance with the acceptance criteria set in section 4.4. Further analysis and investigation of the results in section 6.2 is presented in their corresponding sections. Lastly, possible improvements to the system are proposed and discussed.

7.1 System results

The collection of vibration data from the deployed devices proved that the system worked as intended and thus pass **AC1**. Furthermore, the provided data in table 6.2 shows that the data was received and logged in order. This can be seen by looking at the second value which is the timestamp. Thus the system passes **AC2**. Looking at the same sample, the fourth value is the current battery level, proving that the device provides diagnostic-data. The third value is the local timestamp, if this timestamp is less than the local timestamp in the last received message, the end-device has restarted. This can be used by the server to detect if an end-device has reset. Also if an end-device hasn't been heard from in more than 15-minutes, it can be assumed that said device is either dead or out of reach. This is in accordance with **AC10**. During transmission the device had different LED's toggled, this was confirmed visually and is in accordance with **AC13**. Furthermore in the section 5.2 it can be seen that headers for peripherals have been provided. Some of these have been used for debugging, this is in accordance with **AC14**.

A screenshot of the results from the calibration of one device is provided in fig. 6.25. This shows that the accelerometer offset is properly set. However, the accelerometer is not calibrated to have equal values relative to each other. This means that a measured value on one device may deviate from the same measurement on a different device, this is not ideal. Thus the system fails **AC3**. However, this result also shows that the device indeed was able to provide debug data, thus passing **AC12**.

The end-device was able to synchronize with the RT-server as seen in section 6.2.1, this is in accordance with **AC4** and the limitations stated in section 1.3. However, 2 system configurations have been used to show this. Further testing should be done with a complete network and deployment. The server was able to synchronize with a time-server, this can be seen in section 6.2.5, thus passing **AC5**.

In order to pass **AC6** an end-device must be able to receive and apply configuration parameters. This was done and is shown in section 6.2.7, the criteria are therefore passed. A WDT(watchdog-timer) was utilized in order to handle unexpected errors. However, this was never fully implemented. During transmission, the device turns off the WDT. The reason was that the watchdog would trigger resets at a seemingly random rate during transmission. The cause of the problem was never found thus the system fails **AC7**.

Lastly, each end-device is fully battery-driven, the battery-lifetime can be calculated using the results in section 6.2.2. The battery lifetime expectancy varies with the period between events and applications. The devices deployed averaged between 16 and 20 events per day. Using the worst case of 20 messages a day as the scenario, the total airtime per day is $188.8 \text{ s} \times 20 = 3777.6 \text{ s}$. This result in a total daily(24 h) consumption of:

$$(19.75 \text{ mA} \times 3777 \text{ s}) + (651 \text{ } \mu\text{A} \times 82 \text{ } 623 \text{ s}) = 0.0356 \text{ A h} \quad (7.1)$$

Given that the Samsung INR18650-25R has a capacity of 2.5 A h[14] the expected lifetime can be said to be approximately 70 days. This is a little more than 2 months, and is simply put not a good result. Thus **AC8** is not accepted. A further analysis of the battery-lifetime is provided in section 7.3. The device was however portable and capable of transmitting data from the office to remote gateways using TTN. Thus passing **AC11**. The device was also able to enter and exit sleep mode as seen in fig. 6.18 and fig. 6.17, which is in accordance to **AC9**.

7.2 Synchronization and clock accuracy

This section will cover a deeper analysis of the provided results in section 6.2.1 and section 6.2.6.

7.2.1 End-device clock drift

The end-device clock drift can easily be calculated using the results in section 6.2.6 by using the timestamps in fig. 6.27 and fig. 6.28. The drift can be found by using the following equation ineq. (7.4), the used variables are described in table 7.1.

$$TD_{server} = TE_{server} - TS_{server} \quad (7.2)$$

$$TD_{device} = TE_{server} - TS_{server} \quad (7.3)$$

$$T_{drift} = \left(\frac{TD_{device}}{TD_{server}} \right) - 1 \quad (7.4)$$

TS_{server}	Server time at the start of the test.
TE_{server}	Server time at the end of the test.
TS_{device}	Device time at the start of the test.
TE_{device}	Device time at the end of the test.
TD_{server}	Total time passed from server perspective(actual time passed).
TD_{device}	Total time passed from device perspective (relative time passed).
T_{drift}	The clock drift for the device.

Table 7.1: Table describing the different identifiers in eq. (7.4)

The clock drift can then be calculated using the numbers from section 6.2.6 in eq. (7.4):

$$T_{drift} = \left(\frac{2\,688\,737\text{ s}}{2\,687\,343\text{ s}} \right) - 1 = 5.19 \times 10^{-4} \quad (7.5)$$

The end-device clock is thus faster than the server-clock, making it deviate with approximately 1.87 s per hour or 44.8 s per day. Even though no results are presented, the drift of a second device was checked over a smaller period of time. Said device had a different drift than the one tested. Given the requirements for traditional DMS stated in table 2.1, the device must be synchronized within a certain period of time in order to be

usable in a DMS. Since the device is remote and wireless, this calls for the necessity of a OTA-synchronization-scheme.

7.2.2 Synchronization accuracy

The synchronization was tested on the private-network with $DR = 5$ and $DR = 0$. Based on the results stated in section 6.2.1, it was possible to achieve synchronization within ± 223.1 ms with $DR = 5$ and ± 140.4 ms with $DR = 0$. However as seen in both fig. 6.8 and fig. 6.12, there is a constant positive deviation, so improvements can be made in order to lower this deviation. Comparing this result with the result in section 2.1, there is a large difference in the result. There are a lot of potential error-sources, like the gateway which runs an instance of *Linux* and a *NodeRED* application. These are not RT-compliant in the least were as [2] used all embedded hardware. So the deviation can be somewhat justified, furthermore the system in fig. 5.20 has the potential to be deployed over a quite larger area.

Looking at fig. 6.5 and fig. 6.9 it is clearly spikes were the synchronization is way off. This is due to sync-messages(uplinks) being re-sent, this should be addressed in the future, as the spikes are quite severe in proportion. Looking at fig. 6.5 again, the deviation makes a sudden "jump" at message 70. This is because of the gateway configuration. The gateway is configured to hold 1 downlink in the downlink-queue, which is the least amount of messages it can hold. This causes a constant deviation upon uplink re-transmissions. If an uplink is sent and received but the confirmation sent to the end-device is never received. The uplink is still forwarded to the server, and it will create a callback with a timestamp. However, since the end-device never received a confirmation it will initiate a new message. In the meantime, the callback has been made. The downlink corresponds to an earlier message, causing the timestamp to shift approximately the time between the first and second messages. This should be addressed as it makes the synchronization-scheme quite unstable. Furthermore, the synchronization was tested using only one device. How the scheme holds up with several devices, is not known.

7.3 Battery lifetime expectancy

As earlier stated in section 7.1 the estimated battery-lifetime of an end-device was estimated to be 70 days. One of the biggest factors here was something that can be considered as a bug on the RN2843. A post[26] written June 2019 was found on a forum-site, it stated that

others had the same issue. So according to the LoRaWAN-parameters[4] the maximum payload length varies with the Data-rate. The limitations are 51 B for $DR = 0$ and 222 B for $DR = 5$. The RN2483 is programmed to act according to these limitations and return a *invalid_data_len*-response whenever these are violated. However, after a random amount of messages sent, the RN2483 enforces that 51 B is the maximum payload length for all data rates. Whenever this happens, no more messages can be sent from the RN2483 before it has been fully reset. This was solved by making the max payload size 51 B, however, this increased the number of sent messages from 5 messages to 21 for each event. In addition to this, each message is sent as a confirmed message. This is to guarantee that it is received by the . A lot of the messages fail to transmit the first time. Such that the actual amount of messages sent is often surpassing 30. This can be seen in fig. 6.14 where 30 messages were sent during 1 event. This consumes considerably more energy and contributes to unnecessary large consumption. The consumption without sending any event-messages has been calculated in eq. (7.6). The battery-time is then 160 days when sending messages every 15-minutes. This is not even close to the theoretical estimations given by [6] in section 2.3. However, the device should have the potential for a longer battery-lifetime. If the system could be made such that it accepts message loss and maybe tested with a different LoRa-transceiver, it could see an improvement in battery-lifetime.

$$(651 \mu\text{A} \times 86\,400 \text{ s}) = 0.0156 \text{ A h} \quad (7.6)$$

As a simple end-device

The device was also tested as a simple end-device. In this mode the end-device sends unconfirmed messages and with the accelerometer set at a 1.25 Hz sampling-rate. Given that the payload is 10 B, using eq. (3.12) the air-times are 991.23 ms for $DR = 0$ and 46.34 ms for $DR = 5$. Given that a device is on air approximately 30 s each day, the total amount of messages sent with $DR = 5$ will be $\frac{30 \text{ s}}{46.34 \text{ ms}} \approx 647$. Using the results from section 6.2.2 the daily consumption can be calculated as:

$$647 \times 2.47 \text{ s} \times 7.64 \text{ mA} + 84\,801.91 \text{ s} \times 326 \mu\text{A} \approx 11.1 \text{ mA h} \quad (7.7)$$

This with a theoretical battery capacity of 2 A h, the battery-life is approximately 180 days. Furthermore if the data-rate was configured to $DR = 0$ the amount of messages would be

$\frac{30 \text{ s}}{991.23 \text{ ms}} \approx 30$. This results in a actual daily consumption of:

$$30 \times 3.85 \text{ s} \times 20.1 \text{ mA} + 86\,284.5 \text{ s} \times 326 \mu\text{A} \approx 8.45 \text{ mA h} \quad (7.8)$$

With the same theoretical battery capacity of 2 A h, the battery-lifetime can be calculated to approximately 236 days.

Comparing these results still not in accordance with the theoretical result provided in section 2.3, however, these are much better estimates than that of the deployed devices. Given that the minimum power consumption was measured to 326 μA , there must be something that drains power. Some of the sources to this are known, such as the battery-circuit and the light-sensor. However, these are not solely responsible for the drain. Further investigation is needed to identify the sources responsible for this drainage.

7.4 Vibration data analysis

The collected data from the 3 devices can be seen in section 6.2.3. The devices were set to be triggered at 4mg, that means that device 1 and 2 starts recording at the moment the door is pushed/pulled on. Device 3 is triggered whenever the door-handle is operated. Devices 1 and 2 were deployed on the same surface thus the data from these two should correlate. This can be seen in fig. 6.20 - fig. 6.22. Since the X-axis is the vertical axis, the forces can be seen as very subtle. This is because the device is only exposed to vibration at the actual moment the door is opened and slammed shut. Studying fig. 6.20 this can be clearly seen at Device 2, the first spike is subtle were as the second spike is quite large. This is from the slamming of the door and the oscillations that goes through it afterward. From fig. 6.3 it can be deduced that the Z-axis on devices 1 and 2 should measure the largest force, this can be confirmed by examining fig. 6.22. The most force detected was around 0.1g on device 2, this is to be expected due to the placement of the device. By looking at the spikes in fig. 6.22, it can clearly be seen when the door was opened and closed, especially for the data of device 2. A clear positive force can be seen followed by a halt, then it lingers for about 2 seconds before a clear negative force and some vibrations. The first spike is from opening the door, the halt is from the person walking through the door, and finally, the last spike is from the closing motion. Furthermore, a clear correlation can be seen between devices 1 and 2, in all the plots. The data aligns very well even though they are only timestamped by the server, not by the end-devices. Looking at the range of forces recorded, the sensitivity

should have been increased for both devices.

Looking at the plotted data for device 3 in fig. 6.23 it can be seen that a swift distinct vibration is detected after 0.8 seconds. This vibration comes from the lock cycling upon opening the door. The door has a spring-loaded door lock which cycles every time it is opened or closed. So device 3 is triggered by the door handle, then after 0.8 seconds the door is pulled open. This data does not correlate well with the data from device 1 and 2 since it starts recording before they are triggered.

The devices were able to provide readable data, which do give information about their environments. The data was aligned well, even though the devices were not synchronized. The devices were never deployed out in the real world, only locally. A full-fledged deployment should be done in order to test the system further and to collect more valuable data. Also, an investigation if time-stamping messages combined with OTA-synchronization could improve the accuracy of the system. The scheme is not proven to hold up if the number of devices increases, thus a larger more comprehensive investigation is needed.

Finally a plot of the recorded light levels for device 1 has been plotted in fig. 6.24, this is not directly a part of the assignment. However, as seen in the plot, the light varies throughout the day according to the day-light. The dip in the plot at 11:00 is due to the office being vacant. There are automatic lights in the office. This proves that such a device can gather several different kinds of information.

7.5 Areas of improvement

Battery life

As earlier stated, battery life of 70 days is not adequate. Calculations show that the device may have a theoretical battery-life expectancy between 180 and 230 days dependant on the data-rate used. Even though this is an improvement, more could be done. The circuit could be designed better, using transistors to turn parts of the circuits on and off upon use. Furthermore another RF-transmitter could be used. Because of the bug described in section 7.3, the device ends up sending up to 6 times more messages than necessary. Switching out the RN2483 with another transmitter might do the trick. The RN2483 can also take direct radio commands skipping the mac-layer, so this might be a solution. This, however, requires a much more complex driver. The Atmega324PB is not specialized for

low-energy solutions, neither is the MPU6050, there might be better options. That being said, an ultra-low power design was never the intent. The intent was to create a DMS using LoRaWAN for communication, which has been done.

Full system deployment

As of today, the deployment has been done in two parts. Using the TTN as a service provider and with a private network. Further development of the private network can be done, implementing a server that can be used to synchronize and as an MQTT-broker. In addition, a more elegant and dynamic way of applying configuration parameters could be implemented. This would make a more complete end-to-end solution. Also, the synchronization scheme needs more work. As discussed in section 7.2, the scheme has some bugs, causing a constant deviation due to the en-queuing of messages. A better more stable solution is needed in order to deploy a stable system. Improvements could also be done in terms of synchronization accuracy as well. Lastly, the system should be deployed and tested in the field. This could be done by deploying devices along-side railway-bridges or similar. Data collected from such a deployment could, for instance, be used for analysis to investigate structural integrity, and could prove valuable. This was the intent from the start but never came to fruition.

End device

In addition to the hardware design mentioned in section 7.5, the end device needs more work with the software. The end-device is currently not able to recover from unexpected errors, so a better WDT implementation is advised. Furthermore, the RTC-clock drift is quite big, reducing this would also be preferable. A solution is to implement an actual RTC-module, this would improve both accuracy and SW complexity. Furthermore improving the total amount of memory would be ideal, the current solution is to use an internal FIFO-buffer on the MPU6050 to store accelerometer-data. In order to record longer time-series with more resolution, more memory is needed. This could also enable the device to record data throughout an entire event. And after, when idle, send a large amount of data continuously. Furthermore, a better calibration scheme should be implemented, as is the calibration only sets the offset. However, the sensitivity and actual measured force are not calibrated. A way to measure and set a constant gain, in order to calibrate all devices to measure the exact same amount of force is advisable. The results presented in this paper might not be exact

measurements.

Conclusion

A full-fledged DMS designed to collect vibration-data has been suggested and implemented. The thesis has proved that such a system is able to remotely collect vibration-data and send said data to a server using LoRaWAN as infrastructure.

An embedded design has been proposed and implemented as a vibration-sensor. The design was realized by ordering several assembled boards, which were calibrated and deployed to collect different types of vibration-data. During the deployment, an investigation was done to estimate battery-lifetime expectancy and synchronization properties. The system has been proved to work and was able to collect vibration data which were stored on a server as .csv files.

Most of the acceptance criteria were satisfied. Most of the failed criteria were not of vital importance for the complete solution and resulted in a lower-battery lifetime and less precise vibration measurements. The device was never able to fully handle unexpected errors, making the system more fragile and prone to errors.

Even though a full-solution was not deployed the different concepts and use-cases were proven to work.

Chapter 9

Further Work

The following list of suggested work is based on the discussion in chapter 7.

Suggestions

- A re-design/improvement of the end-device HW to improve battery life.
- Reduce the number of messages needed for the end-device to transmit data.
- Improve the RTC-accuracy to get a more reliable clock on each end-device.
- Improve the calibration scheme in order to get consistent measurements across end-devices.
- Improve the reliability and accuracy of the synchronization-scheme.
- Create a single complete solution implementing both data-logging and end-device synchronization.
- Find a suitable application and deploy the system on a larger scale.

Bibliography

- [1] M. Rizzi, P. Ferrari, A. Flammini, and E. Sisinni, "Evaluation of the IoT LoRaWAN Solution for Distributed Measurement Applications," *IEEE Trans. Instrum. Meas.*, vol. 66, no. 12, pp. 3340–3349, dec 2017.
- [2] L. Tessaro, C. Raffaldi, M. Rossi, and D. Brunelli, "Lightweight Synchronization Algorithm with Self-Calibration for Industrial LORA Sensor Networks," in *2018 Work. Metrol. Ind. 4.0 IoT, MetroInd 4.0 IoT 2018 - Proc.* Institute of Electrical and Electronics Engineers Inc., aug 2018, pp. 259–263.
- [3] M. Kuzlu, M. Pipattanasomporn, and S. Rahman, "Communication network requirements for major smart grid applications in HAN, NAN and WAN," pp. 74–88, 2014.
- [4] LoRa Alliance, "LoRaWAN 1.1 Regional Parameters," LoRa Alliance, Inc., Tech. Rep., 2017. [Online]. Available: https://lora-alliance.org/sites/default/files/2018-04/lorawantm_regional_parameters_v1.1rb_-_final.pdf
- [5] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the Limits of LoRaWAN," in *IEEE Commun. Mag.*, vol. 55, no. 9. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 34–40.
- [6] M. Costa, T. Farrell, and L. Doyle, "On energy efficiency and lifetime in low power wide area network for the Internet of Things," in *2017 IEEE Conf. Stand. Commun. Networking, CSCN 2017.* Institute of Electrical and Electronics Engineers Inc., oct 2017, pp. 258–263.
- [7] e. L. A. T. Committee, "LoRaWAN™ 1.1 Specification," p. 101, 2017. [Online]. Available: https://lora-alliance.org/sites/default/files/2018-04/lorawantm_specification_v1.1.pdf
- [8] Semtech Corporation, "Datasheet SX1276/77/78/79 LoRa Transcieverl Semtech," 2019. [Online]. Available: https://www.semtech.com/uploads/documents/DS_SX1276-7-8-9_W_APP_V6.pdf

- [9] T. U. Rasmussen, "TTK4550 - Use of LoRaWAN for IoT communication , Including Investigation of Connectivity Properties ." no. December, p. 60, 2018.
- [10] "The Things Network," last visited 2020-02-05. [Online]. Available: <https://www.thethingsnetwork.org/>
- [11] Microchip Technology Inc., "RN2483 LoRa™ Technology Module Command Reference," 2015. [Online]. Available: <https://ww1.microchip.com/downloads/en/DeviceDoc/40001784B.pdf>
- [12] —, "RN2483 LoRa™ Datasheet," pp. 1–22, 2017. [Online]. Available: <http://ww1.microchip.com/downloads/en/devicedoc/50002346c.pdf>
- [13] S. Baksa and W. Yourey, "Consumer-Based Evaluation of Commercially Available Protected 18650 Cells," *Batteries*, vol. 4, no. 3, p. 45, sep 2018. [Online]. Available: <http://www.mdpi.com/2313-0105/4/3/45>
- [14] Samsung, "INR18650-25R datasheet," p. 16, 2014. [Online]. Available: <https://datasheetspdf.com/parts/INR18650-25R.pdf?id=839321>
- [15] I. M. Technology, "MCP73830/L Datasheet," p. 22, 2014. [Online]. Available: <http://ww1.microchip.com/downloads/en/DeviceDoc/20005049D.pdf>
- [16] I. Inc., "MPU-6000 and MPU-6050 Product Specification Revision 3.4," p. 52, 2013. [Online]. Available: <https://43zrtwysvxb2gf29r5o0athu-wpengine.netdna-ssl.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf>
- [17] ON Semiconductor, "NCP718 Datasheet," pp. 1–10, 2018. [Online]. Available: <https://www.onsemi.com/products/power-management/dc-dc-controllers-converters-regulators/ldo-regulators-linear-voltage-regulators/ncp718>
- [18] Microchip Technology Inc., "ATmega324PB Datasheet," 2017. [Online]. Available: <http://ww1.microchip.com/downloads/en/DeviceDoc/40001908A.pdf>
- [19] "mkschreder/avr-ultimate-driver-pack," last visited 2020-01-24. [Online]. Available: <https://github.com/mkschreder/avr-ultimate-driver-pack>
- [20] "Arduino String Data type," last visited 2020-01-25. [Online]. Available: <https://www.arduino.cc/reference/en/language/variables/data-types/string/>
- [21] I. Inc., "MPU-6050 Register Map and Descriptions," pp. 1–46, 2013.

BIBLIOGRAPHY

- [Online]. Available: <https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf>
- [22] —, “MPU-6500 Register Map and Descriptions,” pp. 1–47, 2013. [Online]. Available: <https://www.invensense.com/wp-content/uploads/2015/02/MPU-6500-Register-Map2.pdf>
- [23] Multitech, “MultiTech Conduit ® IP67 Base Station IP67 Conduit for Outdoor LoRa ® Deployments EU868 for Europe.” [Online]. Available: <https://www.multitech.com/documents/publications/data-sheets/86002215.pdf>
- [24] Z. Hudak, “UIPEthernet - mbed library for ENC28J60 Ethernet modules. Ful... | Mbed,” last visited 2020-01-30. [Online]. Available: <https://os.mbed.com/users/hudakz/code/UIPEthernet/>
- [25] T. T. Network, “Python SDK,” last visited 2020-02-02. [Online]. Available: <https://www.thethingsnetwork.org/docs/applications/python/>
- [26] “RN2483 - Invalid data length | Microchip,” last visited 2020-02-08. [Online]. Available: <https://www.microchip.com/forums/m1100488.aspx>

Appendix A: The Things Network

A1: Application server GUI

The screenshot displays the Things Network application server GUI, organized into several sections:

- APPLICATION OVERVIEW**: Shows the application ID as `vibration_measurement`, description as "Master thesis: Collecting vibration data", created 20 days ago, and handler as `ttn-handler-eu` (current handler). A [documentation](#) link is available.
- APPLICATION EUIs**: Shows a list of EUIs with a search icon and a [manage euis](#) link. One EUI is visible: `70 B3 D5 7E D0 02 8E 22`.
- DEVICES**: Shows 4 registered devices. A [register device](#) button and a [manage devices](#) link are present.
- COLLABORATORS**: Shows a collaborator named `tobulf` with a profile icon. Action buttons for `collaborators`, `delete`, `devices`, and `settings` are available. A [manage collaborators](#) link is also present.
- ACCESS KEYS**: Shows two access keys: `default_key` and `mqtt_broker`. Each key has buttons for `devices`, `messages`, and `settings`. The key values are displayed in a masked format (dots) with a `base64` label and a copy icon.

A2: Device control panel

DEVICE OVERVIEW

Application ID `vibration_measurement`

Device ID `office_test_1`

Activation Method `OTAA`

Device EUI `<> 00 E1 1F BD 2B 01 E7 31`

Application EUI `<> 70 B3 D5 7E D0 02 8E 22`

App Key `<>`

Device Address `<> 26 01 2B DE`

Network Session Key `<>`

App Session Key `<>`

Status ● 6 hours ago

Frames up 11600 [reset frame counters](#)

Frames down 15303

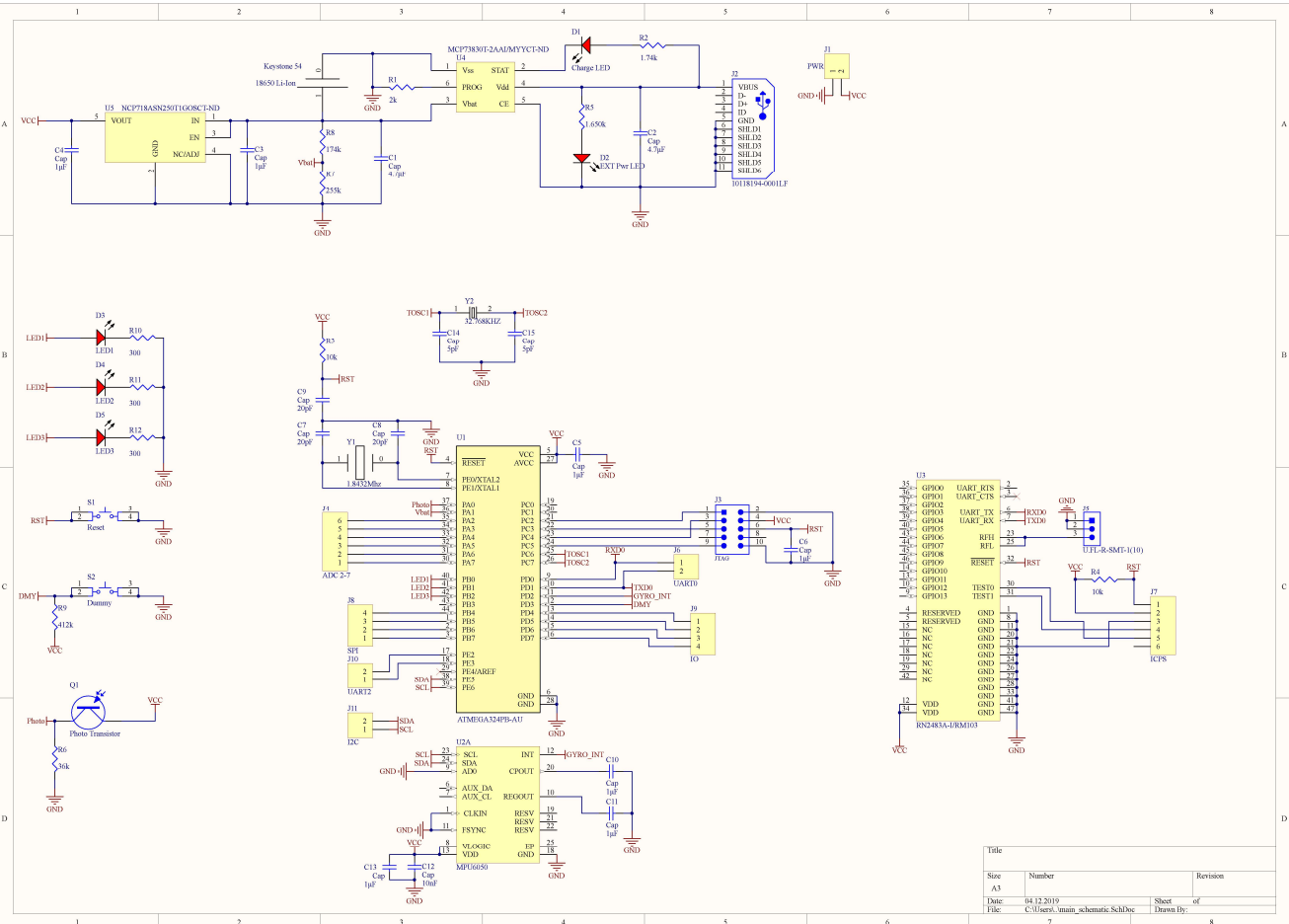
DOWNLINK

Scheduling `replace` `first` `last` FPort `1` Confirmed

Payload `bytes` `fields` `0 bytes`

Appendix B: HW End-device

B1: Schematic

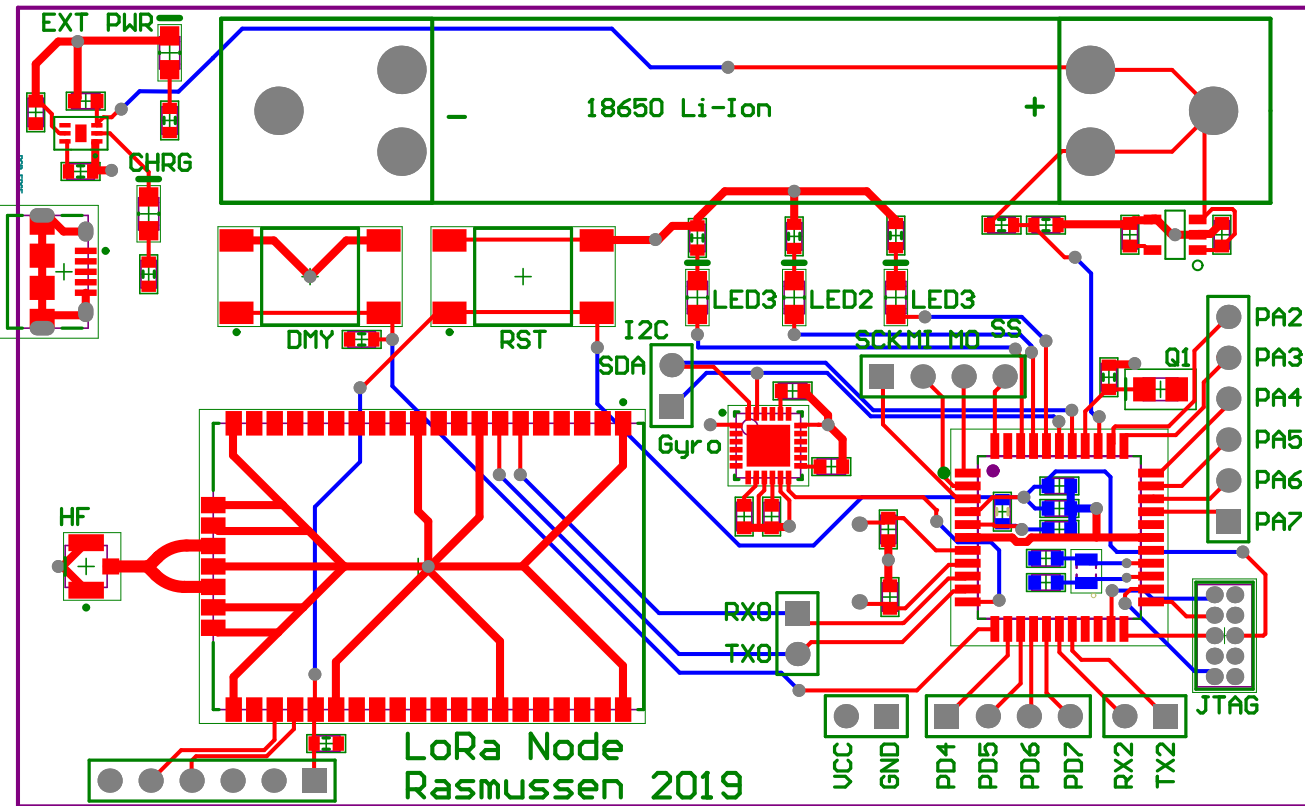


Size	Number	Revision
A3		
Date:	04.12.2019	Sheet of
File:	C:\Users\mnam_schematic\Schloc	Drawn By:

B2: Part list

Designator	Description	Value	Package
18650 Li-Ion	Battery Clip 18mm cell	-	Through Hole
C1, C2	SMD Capacitor	4.7 μ F	SMD 0603
C3-C6, C10, C11, C13	SMD Capacitor	1 μ F	SMD 0603
C7, C8, C9	SMD Capacitor	20 pF	SMD 0603
C12	SMD Capacitor	10 nF	SMD 0603
C14, C15	SMD Capacitor	5 pF	SMD 0603
D2, D3	LED, Red	-	SMD 0805
D1, D4	LED, Yellow	-	SMD 0805
D5	LED, Green	-	SMD 0805
J1, J6, J10, J11	Board-To-Board Connector	-	Through Hole
J2	Micro USB Connector	-	SMD
J3	Wire-To-Board Connector	-	Through Hole
J4, J7	Board-To-Board Connector	-	Through Hole
J5	Coaxial Connector, MCX	-	SMD
J8, J9	Board-To-Board Connector	-	Through Hole
Q1	Phototransistor	-	SMD 0805
R1	SMD Chip Resistor	2 k Ω	SMD 0805
R2	SMD Chip Resistor	1.74 k Ω	SMD 0805
R3, R4	SMD Chip Resistor	10 k Ω	SMD 0805
R5	SMD Chip Resistor	1.65 k Ω	SMD 0805
R6	SMD Chip Resistor	40 k Ω	SMD 0805
R7	SMD Chip Resistor	255 k Ω	SMD 0805
R8	SMD Chip Resistor	174 k Ω	SMD 0805
R9	SMD Chip Resistor	412 k Ω	SMD 0805
R10, R11, R12	SMD Chip Resistor	300 Ω	SMD 0805
S1, S2	Tactile switch	-	SMD Button
U1	8 Bit MCU, AVR Family	Atmega324PB	44-pin TQFP
U2	Motiontracking MEMS	MPU6050	24-pin QFN
U3	RF Transceiver	RN2483	RM
U4	Battery Charger	MCP73830T	2x2 TDFN-6
U5	LDO voltage regulator	NCP718ASN250	TSOT 23 5
Y1	Crystal	1.8432 MHz	Through Hole
Y2	Crystal	32.768 kHz	SMD

B3: PCB trace



Appendix C: Private gateway

C1: Gateway LoRa configuration

LoRa Mode

Mode	Packet Forwarder	Network Server	Lens Server
NETWORK SERVER	4.0.1-r19.0	2.2.30	2.2.30
Restart LoRa Services	Status	Status	Status
	RUNNING	RUNNING	DISABLED

LoRa Card Information

Gateway EUI	00-80-00-00-A0-00-33-BB
Frequency Band	868
FPGA Version	31

LoRaWAN Network Server Configuration

Channel Plan

Channel Plan	Additional Channels 1 (MHz)	Duty Cycle Period (min)
EU868	867,5	60
Channel Mask	Edit	

Network

Network Mode	Join Delay (sec)	Lease Time	Address Range Start
Public LoRaWAN	5	00-00-00	00:00:00:01
NetID	Rx1 Delay (sec)	Queue Size	Address Range End
000000	1	1	FF:FF:FF:FE

Settings

Tx Power (dBm)	Rx 1 DR Offset	ADR Step (cB)	Min Datarate
26	0	30	0 - SF12BW125
Antenna Gain (dBi)	Rx 2 Datarate	ACK Timeout	Max Datarate
3	0 - SF12BW125	5000	5 - SF7BW125

C2: Gateway join server





Join Server

Location

Local Keys

Local End-Device Credentials

[Add New](#)

Device EUI	App EUI	App Key	Class	Device Profile	Network Profile	Options
00-23-A1-EF	00-23-A1-EE	****F113BC4A	A	LW102-OTA-EU868	DEFAULT-CLASS-A	 
00-EB-9F-66	00-23-A1-EE	****F113BC4A	A	LW102-OTA-EU868	DEFAULT-CLASS-A	 

Local Network Settings

 Enabled

Network ID (AppEUI)

EUI

EUI

0004A30B0023A1EE

Default Profile

Network Key (AppKey)

Key

DEFAULT-CLASS-A

Key

93ADE996B818DE123617D6