NTNU

Institutt for elektroniske systemer

Bacheloroppgave

Laveffekts IoT Smartsystem IoT based low power sensor system.	Gitt dato: 08.01.2020 Innlevingsdato: 29.05.2020 Antall sider/bilag 168
Harald Sæther Kåre-Benjamin Hammervold Rørvik Runar André Saure Tom Kristian Moen	Veileder (navn/email): Arne Midjo arne.midjo@ntnu.no
Elektroingeniør, Elektronikk	Prosjektnummer: E2026
NTNU med Royal Cloud Solutions	Kontaktperson hos oppdragsgiver (navn/tlf.): *Arne Midjo* *73412662*

Fritt tilgjengelig

X

Preface

This report was written during the spring semester of 2020. The report has a scope equivalent of 20 credits and was made in cooperation with NTNU. The thesis is a result of the bachelor project *IoT based low power sensor system* and is the final part of the team members electrical engineering study at NTNU Trondheim.

The IoT subject was introduced during the autumn semester of 2019 in the electrical engineering study in which the team members attended. As the subject was highly interesting, the team members decided on pursuing IoT as part of the upcoming bachelor. Such was the bachelor project *IoT based low power sensor system* born.

The government restrictions due to the Covid-19 pandemic did not discourage the teams effort to make a fulfilling bachelor project. Firstly, the supply shortage of early 2020 delayed the initial prototype manufacturing from early January until March. Secondly, a total faculty lockdown came into effect at 12th of March 2020. The team restructured the scope of the thesis accordingly and made it increasingly theoretical.

We would like to express our gratitude towards NTNU for economically supporting us, providing experienced employees and tools. Omega Workshop NTNU for hardware assembly tools and parts. Amund Askeland for the technical support, expertise, and devices. Arne Midjo for his assistance, guidance, and supportive attitude. He enabled us to take on this complex project and was crucial for a successful execution amidst the covid-19 chaos.

Without the help and support of these players, there would not be a project with enough satisfactory professional weight.

Trondheim, May 2020

Abstract

A wireless low power sensor system reading changes in a building environment. Using a database and dynamic wireless node structure to transmit and provide remote clients with a reliable, secure, visual, and user-friendly overview of their data and room state.

Summary

The background for the task explored during the bachelor project is the employer (NTNU)'s desire to support the research and development of a system capable of monitoring simulations events in multiple rooms. The system is urged to feature encrypted and robust wireless transmissions between modules, and a system with recorded data stored securely in a remote database, where relevant data for the room state may be retrieved and displayed through a web browser or on a mobile phone.

Similar systems such as fire alarm or motion detection are systems the team have taken inspiration from, while developing the low power IoT sensor system. By using the wireless connectivity and low power capabilities of already existing MCUs the team connect simple sensors that can monitor different values inside a closed space. Using Wi-Fi for internet access and a cloud-based database. The data from multiple rooms can be stored securely and accessed through a web browser on a PC or mobile phone. The team connect a low power MCUs wirelessly to a collector, which uplinks the data to a cloud-based database, where the data can be accessed. This data is further downlinked to a server, which interfaces with a web browser through a secured socket. This creates a wireless sensor system, where the data stream goes from the low power MCUs, also called nodes, to a collector/gateway. Which is connected to an uplink that connects the physical nodes, gateway, and uplink to the internet where the data ends at the end clients web browser.

With identifiers the network can have multiple nodes and the gateways, and if a user of the system knows which identifier corresponds to which door the user can observe every door on a floor or in a building simultaneously. Having such a system means that a security guard does not need to manually check every door on a commercial building, to make sure they are closed, only needing a web browser to see which doors are open. Nodes achieving industry lows in terms of power consumption, years can go by without needing a battery change.

Table of contents

Introduction

The public value of incorporating smart solutions into office spaces, homes and buildings is increasingly emphasized to engineers and system designers. New smarter solutions are most likely to replace or work in conjunction with older routines, such as manual checkups, and or lack thereof. There are various systems and features that already incorporate smart homes and solutions. However, they are largely perceived by the public (i.e. the end user) as pioneering solutions geared towards younger users and early adopters.

This project aims to explore and improve upon existing systems, focusing on a low-cost and easy-to-use system. This will be achieved by making a new system from scratch, exploring the design process and the viability of combining already existing IoT-technologies. This includes concepts such as low power, wireless communication, secure transmissions, ease of use, quick response times and simplicity.

The design team values innovation and smart solutions, with a focus on being part of the ever-developing field of IoT, helping pioneer and define what one can associate with future IoT system.

The thesis is structured by the Standard IEEE format, meaning that the thesis will first present the theory followed by method and the results themselves. Next is discussion, consulting the results achieved, followed by conclusion, sources and appendix.

0 Abbreviations and glossary

Abbreviation	Definition
IoT	Internet of Things
LBT	Listen Before Talk
AFA	Adaptive Frequency Agility
ERP	Effective Radiated Power
CCS	Code Composer Studio
SCS	Sensor Controller Studio
VSC	Visual Studio Code
API	Application Programming Interface
UART	Universal Asynchronous Receiver/Transmitter
PCB	Printed Circuit Board
SMD	Surface Mounted Device
THT	Through Hole Technology
MCU	Microcontroller Unit
IC	Integrated Circuit
I^2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
LED	Light Emitting Diode
ADC	Analogue Digital Converter
ESR	Equivalent Series Resistance
GFSK	Gaussian frequency shift keying
JSON	JavaScript Object Notation
FSPL	Free Space Path Loss
IDE	Integrated Development Environment
BNT-1	Button 1 on the LAUNCHXL-CC1312R launchpad
CPU	Central processing unit
SDK	Software development kit
RF	Radio frequency
Backend	Refers to the backend created with Node.js
Frontend	Refers to the frontend created with HTML
dio	Digital input output

1 Theory

1.1 The Node and the Gateway

The system needing to read multiple changes in different rooms, would employ a gateway to multiple node communication. The idea is not new, where one gateway would communicate with different nodes which send sensor data of the building environment.
The term node is used referring to hardware at the end point of the system, mounted at the room entry point, collecting entry point data and/or room data through sensor mounted on the hardware, along with communicating with the gateway. Sleepy nodes refer to the node's CPU spending most of the time in a low power state.
The term gateway refers to hardware collecting data from all the nodes in the network and sending this collected data further to the uplink hardware, a module capable of communicating with a database unit.

1.1.1 Wireless communications technology

When considering wireless technology for an IoT low power sensors system, it is important that all the system requirements are met by the technology selected. Some common system requirements include:

- Range - The system may be installed in a large house, housing complex, office, or facility building. Therefore, the technology must support a wide area of coverage.
- Low power - for battery operated sensors, it is critical that the wireless technology supports low power connection. Achieving long life is optimal for battery powered IoT-sensor networks, not just because of the hassle of changing batteries on each node, but because it is a security and safety risk.
- Security - The wireless technology should provide some form of protection against security attacks. Intruders might exploit the weakness of the system and break in undetected. When the security network is connected to the internet through a home Wi-Fi network, hackers can use this weakness and attack other connected elements in the house such as door locks.

- Robustness - The connection should be robust against interference, jammers, and different RF conditions. Jamming could be generated by an innocent device but also intentionally by an intruder.
- Scalable - The wireless technology should be scalable so that more devices can be added to the network. A typical security system monitors all entry points in addition to other security measures, in addition to safety sensors such as fire detectors. The number of end nodes in the system in a house can easily reach over 30 nodes. In commercial buildings that number of nodes may be much higher.
- Network Type - one way or two-way communication. Many legacy wireless sensor networks support only one-way communication. They transmit their indication to a central unit without receiving an acknowledgement and without any capability to receive any downlink messages. In advanced systems, there are many use-cases that require two-way communication where the central unit also transmits messages to the end nodes. Such use-cases include simple commands like enable/disable, firmware upgrade, configuration settings and more.

Sub 1 GHz wireless technology answers all the above requirements and is widely used across security systems in buildings thanks to its excellent RF performance, its low power, and its low cost. RF signals in Sub 1 GHz frequency bands propagate well in air, through walls and around corners. Therefore, it is easy to achieve robust wireless signal coverage of an entire house, a large floor, or a building.

1.1.2 TI SimpeLink sub 1 GHz

SimpeLink sub 1 GHz refers to Texas Instruments Sub 1 GHz solution for multiple of their MCUs where the radio transceiver can send signals with a frequency beneath 1 GHz. Texas Instruments boasts about their wireless MCUs with long range and low power[1].

When using sub 1 GHz radio, there are certain requirements in which radio frequency one uses. In Europe 863 MHz to 870 MHz band has been allocated for license free operation using Frequency hopping spread spectrum, direct sequence spread spectrum, or an analogue modulation with either a transmission or duty cycle of 0.1%,1% or 10% depending on the band. See etsi en 300 220-2 standard for short range devices[2].

A 1% duty cycle means that if a receiver and transmitter is talking with each other, they must not talk for 99 seconds if they talk for 1 second. This means that in an hour they cannot talk

for more than a combined 36 seconds per hour. If the transmitter and receiver need 20ms to send and receive a packet this means that they can send a packet every 2 seconds as long as they do not talk on the 1% duty cycle frequency in this 2 second period.

1.1.3 Network topology Node to Gateway

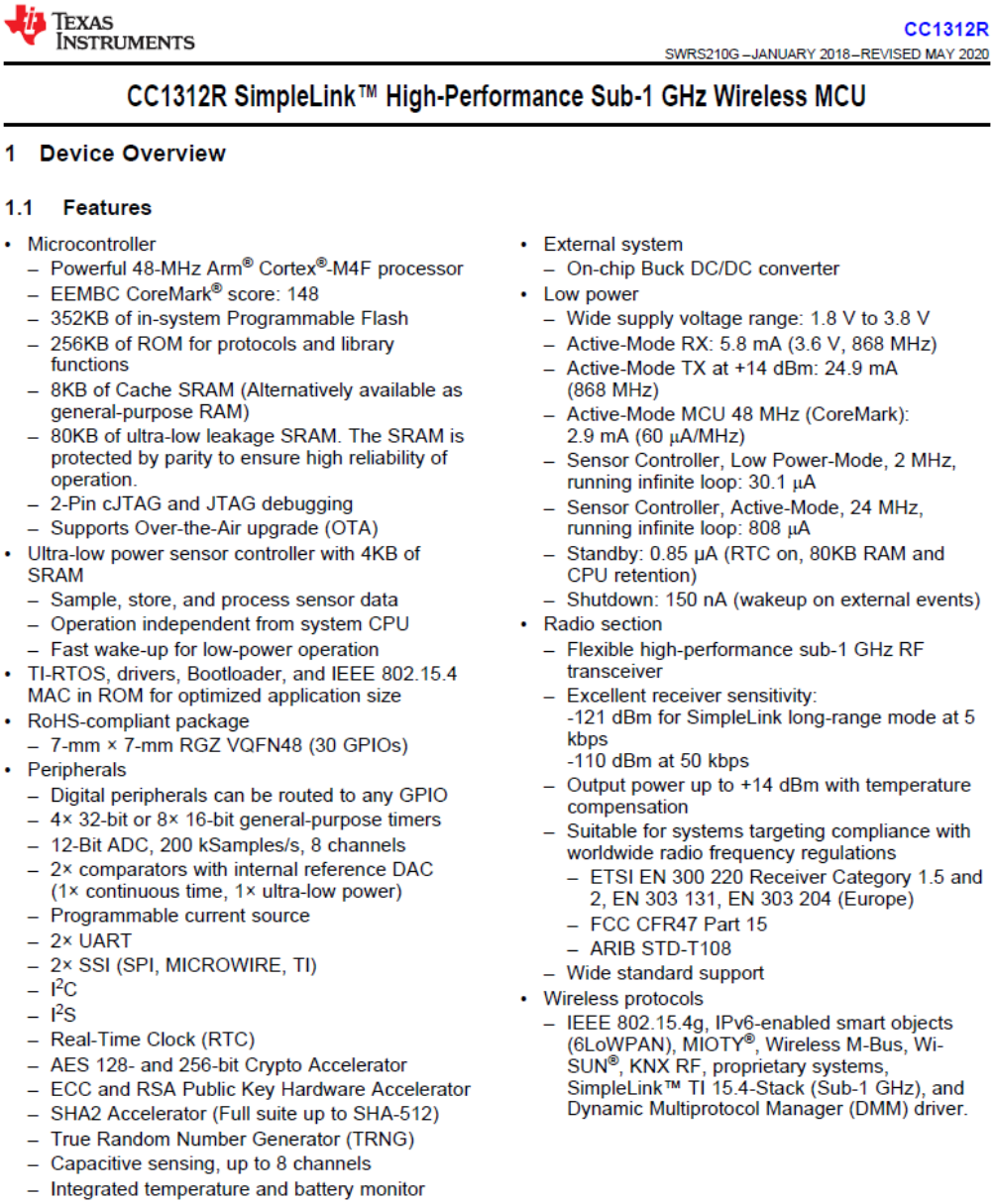
For a system where nodes sleep for most of the time and are connected to one collector, the typical and most feasible network topology is the star network with sleepy end nodes. In star networks, there is one single concentrator/gateway to which all other nodes are connected. Sleepy end nodes mean that the nodes that are connected to the gateway are in their low power state most of the time and wake up to transmit their message periodically or based on event triggers such as the opening or closing of a window or door. Other networks topologies would be:

- Bus network, where the nodes are connected to a common medium along this medium.
- Ring network, where each node is connected to their left and right node neighbour
- Mesh network, where each node is connected to an arbitrary number of neighbour nodes such that there is always one traversal from any node to another.
- Fully connected network, where each node is connected to every other node.
- Tree network, where nodes are arranged in hierarchical order.

1.1.4 CC1312R

The CC1312R MCU[3] from Texas instruments is a member of the CC13x2 and CC26x2 family of cost effective, ultra-low power, 2.4GHz and Sub-1 GHz Rf devices. Very low active RF and microcontroller current consumption, in addition to flexible low power modes, providing excellent battery lifetime and allowing long-range operation on small coin cell batteries and in energy harvesting applications. The CC1312R combines a flexible low power RF transceiver with a powerful 48Mhz Cortex-M4F microcontroller in a platform supporting multiple physical layers and RF standards. A dedicated radio controller (Cortex-M0) handles low level RF protocol commands that are stored in ROM or RAM, thus ensuring ultra-low power and flexibility. The low power consumption of the CC1312R device does not come at the expense of RF performance. The MCU has excellent robustness and sensitivity performance. The chip is a highly integrated, true single-chip solution incorporating a complete RF system and an on-chip DC-DC converter.

With a dedicated autonomous ultra-low power MCU, the sensor controller can handle analogue and digital sensors. Thus, the main MCU (Cortex M4F) can maximize sleep time. The CC1312R power and clock management and radio systems require specific configuration and handling by software to operate correctly, which has been implemented in the TI-RTOS. TI recommends using this software framework for all application development on the device. Texas Instruments recommends using the CCS client as the MCU's code editing software. TI-RTOS, CCS and the device drivers are all free of charge. Features from the CC1312R datasheet can be seen in Figure 1.



TEXAS INSTRUMENTS

CC1312R

SWRS210G – JANUARY 2018 – REVISED MAY 2020

CC1312R SimpleLink™ High-Performance Sub-1 GHz Wireless MCU

1 Device Overview

1.1 Features

- Microcontroller
  - Powerful 48-MHz Arm® Cortex®-M4F processor
  - EEMBC CoreMark® score: 148
  - 352KB of in-system Programmable Flash
  - 256KB of ROM for protocols and library functions
  - 8KB of Cache SRAM (Alternatively available as general-purpose RAM)
  - 80KB of ultra-low leakage SRAM. The SRAM is protected by parity to ensure high reliability of operation.
  - 2-Pin cJTAG and JTAG debugging
  - Supports Over-the-Air upgrade (OTA)
- Ultra-low power sensor controller with 4KB of SRAM
  - Sample, store, and process sensor data
  - Operation independent from system CPU
  - Fast wake-up for low-power operation
- TI-RTOS, drivers, Bootloader, and IEEE 802.15.4 MAC in ROM for optimized application size
- RoHS-compliant package
  - 7-mm × 7-mm RGZ VQFN48 (30 GPIOs)
- Peripherals
  - Digital peripherals can be routed to any GPIO
  - 4× 32-bit or 8× 16-bit general-purpose timers
  - 12-Bit ADC, 200 kSamples/s, 8 channels
  - 2× comparators with internal reference DAC (1× continuous time, 1× ultra-low power)
  - Programmable current source
  - 2× UART
  - 2× SSI (SPI, MICROWIRE, TI)
  - I^2C
  - I^2S
  - Real-Time Clock (RTC)
  - AES 128- and 256-bit Crypto Accelerator
  - ECC and RSA Public Key Hardware Accelerator
  - SHA2 Accelerator (Full suite up to SHA-512)
  - True Random Number Generator (TRNG)
  - Capacitive sensing, up to 8 channels
  - Integrated temperature and battery monitor
- External system
  - On-chip Buck DC/DC converter
- Low power
  - Wide supply voltage range: 1.8 V to 3.8 V
  - Active-Mode RX: 5.8 mA (3.6 V, 868 MHz)
  - Active-Mode TX at +14 dBm: 24.9 mA (868 MHz)
  - Active-Mode MCU 48 MHz (CoreMark): 2.9 mA (60 µA/MHz)
  - Sensor Controller, Low Power-Mode, 2 MHz, running infinite loop: 30.1 µA
  - Sensor Controller, Active-Mode, 24 MHz, running infinite loop: 808 µA
  - Standby: 0.85 µA (RTC on, 80KB RAM and CPU retention)
  - Shutdown: 150 nA (wakeup on external events)
- Radio section
  - Flexible high-performance sub-1 GHz RF transceiver
  - Excellent receiver sensitivity:
    -121 dBm for SimpleLink long-range mode at 5 kbps
    -110 dBm at 50 kbps
  - Output power up to +14 dBm with temperature compensation
  - Suitable for systems targeting compliance with worldwide radio frequency regulations
    - ETSI EN 300 220 Receiver Category 1.5 and 2, EN 303 131, EN 303 204 (Europe)
    - FCC CFR47 Part 15
    - ARIB STD-T108
  - Wide standard support
- Wireless protocols
  - IEEE 802.15.4g, IPv6-enabled smart objects (6LoWPAN), MIOTY®, Wireless M-Bus, Wi-SUN®, KNX RF, proprietary systems, SimpleLink™ TI 15.4-Stack (Sub-1 GHz), and Dynamic Multiprotocol Manager (DMM) driver.

Figure 1: Features from the CC1312 datasheet

1.1.5 Energytrace

Energytrace[4] is a standalone tool for approximating power and amperage consumption along with other information about the CC1312R and other specific Texas instruments MCU's. The tool comes built in with the Texas instrument CCS firmware and has 2 different modes of operation, where it only measures power and amperage consumed by the MCU in a timespan, and where it measures different CPU states.

1.1.6 Battery life calculations of a CC1312R sleepy node:

The main parameters that affect the estimated battery life of a CC1312R sleepy node running on 3.3v CR2032 coin cell battery:

- Battery capacity (As)
- Average standby state current consumption(A)
- Standby state duration(s)
- Average awake current consumption(A)
- awake state duration(s)

$$life = \frac{battery_{capacity(As)}}{\left(\frac{Istandby*tstandby+Ihigh*thigh+Imid*tmid+Ilow*tlow}{ttotal}\right)} * \frac{1}{3600*24*365}$$

Equation 1, approximated battery life

$$life = \frac{battery_{capacity(As)}}{\left(\frac{Istandby*tstandby+Ihigh*thigh+Imid*tmid+Ilow*tlow}{ttotal}\right)} * \frac{1}{3600*24*365}$$

In

Equation 1, ttotal is the time between the events per day, Istandby and tstandby refer to the standby state along with Ihigh and thigh, Imid and tmid, and Ilow and tlow which refer to the wakeup state and duration. The reason for the 3 different high, mid and low, is as in the awake state of the CC1312R chip, the 3 high, mid and low represent average current when the CPU uses a lot of current (high), medium and low current. When the node is awake it goes through 3 different modes, waking up and sending packets, waiting for "ack" from the gateway, and going to sleep. What this

$$life = \frac{battery_{capacity(As)}}{\left(\frac{Istandby*tstandby+Ihigh*thigh+Imid*tmid+Ilow*tlow}{ttotal}\right)} * \frac{1}{3600*24*365}$$

Equation 1, approximated battery life really does, is to find the average current in seconds, finding the number of seconds, and dividing with seconds in a year to get the lifetime in years.

Another way to estimate the battery life expectancy is dividing the day into periods and assigning these periods to "stand by state" and "awake states". The Energytrace tool can record the amount of joules consumed during a period, and can be set to different recording timers, like 10 seconds. If one can capture 1 awake state in a sleepy node one can extract the amount of energy used in this recording of 10 second. Now having a joule amount for an awake state one can find the joules spent per day if one knows the amount of awake states per day. If the standby state is non null, then it also must be taken into consideration. Filling the periods per day the former awake states per day does not have. The amount of joules in the battery divided by the previous value will amount to an approximate battery life. As an example: if there are 2 joules used in a recording of 10 seconds when a reed switch is triggered, and 1 joule used in a recording of 10 seconds when in standby state. The number of joules used per day if there are 64 packets sent will be $2\,J \cdot 64 + 1 \cdot 8576\,J = 8704\,J$. If the amount of joules disposable in the chosen battery is 2362 joules the battery lifetime is: $2362/8704 = 0{,}271\ days$

1.2 Database, frontend and backend.

1.2.1 MongoDB

MongoDB[5] is a free open source NoSQL database server and tools solution. Provided by MongoDB, Inc. on an SSPL (v.1) and Apace(v.2) license[6]. These licenses fully support commercial usage and may thus be used for any purpose. MongoDB supports the following programming languages (without further modifications): C, C#, Java, JavaScript, Perl, PHP and Python

1.2.2 DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service provided by Amazon. It offers a scalable high uptime solution using the AWS hosting structure. DynamoDB is intended to create database tables to store and retrieve data with vast amounts of traffic.

1.2.3 Firebase

Google offers both[7] Cloud Firestore and Real-time Database by default in the standard console. While Cloud Firestore and Real-time Database offer many of the same features, choosing one over the other will prove a difference in feature set and approach. Both solutions offer a cloud-based client-accessible database solution with Real-time data sync support.

Real-time Database is an earlier iteration of Cloud Firestore. Although not outdated, both versions are recommended and still improved and supported by Google (as per 20th May 2020). Real-time Database was originally launched in 2011 by Firebase Inc. and acquired by Google in 2014. Where Cloud Firestore on the other hand is developed by Firebase as an official subsidiary, with the support of the mother company Google. The influences of Google may be noticed upon both products, and especially with the vast feature set expanded upon Real-time Database. Both solutions offer a zero maintenance Client-first SDK which does not require any maintenance nor manually deployed server setup. Both solutions offer a free entry tier, which is quite helpful in development stages. This allows IoT developers to test the console before committing. The two solutions have a different pricing solution beyond the free tier. Real-time Database charges on a strictly bandwidth and storage basis. Where Cloud Firestore charges on a per operation basis, and secondary at a discounted rate for bandwidth and storage. Both platforms support the following languages: C++ Java, JavaScript, Node.js Objective-C, PHP and Swift.

1.2.4 WebSocket

WebSocket is a protocol which enables a two-way communication between a client and a host. The communication is created on a persistent connection where data may be passed omnidirectionally, which is why it is often used for continuous data exchange. Where new data packets may be sent in both directions without breaking connection. The protocol is initiated with a standard opening handshake and is carried out via TCP. The protocol was defined and standardized by IETF [8]. The protocol was intended for browser-based applications in need of two-way communication with servers.

1.2.5 socket.io

Socket.IO is a socket ensuring real-time bidirectional event-based communication. The socket operates much like a WebSocket however is defined by the socket.io (open source collective) and not IETF. Socket.io is a popular solution which is available for frameworks such as Node.js and Arduino IDE[9].

1.2.6 HTTP(s) Protocol

Every HTTP request contains a verb/method at which defines its intention or request, meaning the requests decides the action at which is requested to take place. Some standard HTTP methods includes[10]:

GET - Getting Data. This method may be used to request data from a specified resource. If parts of the information is desired, then a secondary path may be added in place of the OPTIONAL (such as MYNODE). It would then only acquire the information related to MYNODE and not the entire root of the folder (or subfolder).
An example of application could be:

- Request: GET /BRANCH/MYNODE/OPTIONAL
- Response: [{nodeID: 1, humidity: xx, doorState: xx, light: xx, air: xx}]

POST - Creating Data. This method may be used to post data to a specified resource.

- Post /BRANCH/MYNODE

PUT - Updating Data . The Put method may be used to post or overwrite (thus put) data and takes the same form and understanding as the Post method.
DELETE - Deleting Data. The Delete method may be used to delete data from a specified resource, including all branches.

- Delete /BRANCH/MYNODE

1.2.7 CURL

CURL is the tool used to transfer data to the database and supports both HTTP and HTTPS, more information may be found by invoking ‘curl --help’ or ‘curl --manual’.
The -x is similar to the --proxy flag which has the task of overriding environment variables and allows the user to assign a proxy server. As documented and suggested by the Firebase

documentation. The “-d” flag is used in conjunction with the PUT command to imply the transferal of Data and a PUT request.

1.2.8 REST

REST (Representational State Transfer) is a convention for building HTTP services using the already defined and simple HTTP principles. These functions include creating, deleting, reading, and updating data. Combining these letters derives the term CRUD operations. It is a common convention to add a /api in the URL to hint or indicate the usage of a RESTful API. It may also be displayed through a sub domain. The endpoint (also known as resource) is the decisive factor which indicates the requested page with which the client wishes to interact. A popular way of interacting with the endpoint is through HTTP requests and responses, using the HTTP or HTTPS protocol.

1.2.9 ExpressJS

Expressjs is a flexible lightweight framework for Node.js. It offers robust tools for HTTP servers and routing. It is used in conjunction with the built-in functionality within node to create for instance web hosts. It is the 7th most popular npm module and is takes up only 208kb. [11]

1.2.10 Node.js

Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a browser. For simplicity and readability Node.js will be addressed as Node.js and Node during this thesis. Node uses the W8 engine designed for chrome, making it able to run standalone outside of the browser (meaning a web page). The system has a scalable asynchronous event-driven structure, allowing it to flexibly support various fields of usage. The implication of this structure is the allowance of calling several functions, such as call-backs or promises without having to wait for the response. This allows Node to sleep while there is no work to be carried out. Alternatively, Node may complete or launch other tasks while waiting for the former tasks to be carried out. This allows for a Non-Blocking functionality, meaning the system may launch several tasks at once and wait,

instead of locking threads and waiting for completion. Node is typically used in two applications, either front end or back end.
Every file in a node application is considered a module, the variables and functions defined within these files are scoped to said files. As defined in object-oriented programming, as private, being unreachable outside the given container.

Some important modules built into node: File system, HTTP (to create web servers that listens to HTTP requests). OS (to work with the operating system). Path (Gives various utility functions to work with paths). Process (Gives information about the ongoing process). Query Strings (useful in building HTTP services). Stream (Aids in working with streams of data) among many other modules[12].

1.2.11 Postman

Postman is an extension for Chrome which allows the user to perform HTTP methods, such as GET, PUT and POST. A standalone application is also available[13].

1.2.12 Helpful extensions to VScode

Several expansions were used during development to increase the ease of use and productivity. Jshint allows the user to receive auto generated feedback and suggestions for their code. The tool is activated in the terminal by writing: jshint FILENAME.js. The extension then analyses the given code and prompts the user with auto generated feedback. Jshint may be installed using the standard npm syntax “npm install -g jshint”. The -g flag ensures a global installation of the extension to VScode. Warnings and errors may be bypassed by making a custom “.jshntrc” file. One typical application of this feature is ignoring the “const” warning. The warning may be found a nuisance by the coding team, as it is supported by ECMAScript 6 and up. Which in the case of Node.js works well. The warning may be bypassed by adding the following to the “.jshntrc” file. { "esversion": 6 }.ECM JavaScript [14]

1.2.13 PlatformIO

PlatformIO Is a standalone IDE most comparable to the well-known Arduino IDE. PlatformIO is more tailored toward IoT applications and embedded system, while Arduino is a more general microcontroller programmer. It is also included as a plug-in for Visual Studio Code, which is what is used in this project. PlatformIO boasts easy-to-use and intuitive functions, such as easy integration, automatic dependencies fetching, saving the board to the code settings, Git integration, Library manager and more.

1.3 Available Microcontrollers

It is important to give some examples of nodes that were considered during the early phases, giving others a more diverse line-up to choose from if they consider making an IoT-system.

The Arduino Uno has a CPU clock of 16MHz, being a lightweight MCU system, but could possibly be considered too weak for some. The standard model has no Wi-Fi antenna, but a model with an integrated antenna can be purchased at a higher cost, at 451 NOK (~45 USD). A model with no Wi-Fi costs 210NOK (21 USD).

The Espressif ESP8266 is a quite common microcontroller, ranging from industries to hobbyists alike. The 8266 has one CPU core, clocked at 80MHz, but can be adjusted to 160MHz if necessary, at the cost of bigger power draw and higher temperature. It has no included flash memory or SRAM. The ESP8266 is considered the cheapest option, costing 80 NOK (8 USD). The ESP8266 has a large amount of support online, as it is an older module, meaning more people have had time to discover its possibilities and faults.

The ESP32 is a “sized-up” ESP8266, which means that some of the specifications are similar. The ESP32 offers two CPU cores, one which is in control of WI-FI-communication. This means that the second core is free to uninterruptedly process its tasks. The ESP32 can be run at up to 240MHz, a considerable step-up of the 8266. The dual-core CPU and higher speed means that the ESP32 can support a bigger system.

The CC3200, produced by Texas Instrument offers about the same performance as the ESP8266, although with a newer ARM Cortex-M4 processor @ 80MHz. Texas Instrument

has produced large amounts of official documentation, software and support, although it is less used by hobbyists and freelancers. A dev kit is priced at 392 NOK (~39 USD). The CC3200 devkit also has a larger footprint than its competitors.

The Realtek RTL8710 was another consideration, offering many of the same features as the ESP8266, although with a newer single core ARM Cortex-M3 processor locked at 166MHz. In many ways, it is very comparable to the ESP8266, however at a higher price, and is currently struggling with a stock shortage.

The Nufront NL6621 is comparable to RTL8710s specification, utilizing the same CPU but clocked at the slightly lower 160MHz. It does however struggle with the same stock shortage as the RTL8710.

1.3.1 ESP32 Models

Although it seems simple to just choose the ESP32 and be done, one must also consider which ESP32 model one wants to use. There are multiple to choose from, but this short comparison is limited to the "Wroom" model and "Wrover-B" model, both officially manufactured by Espressif.

The main difference between the "Wroom" and "Wrover-B" is the inclusion of 8MB PSRAM in the Wrover-B, which is a variation of regular SRAM, giving it more temporary storage but at a small price increase. There are also small changes in clock speeds, but this has no effect on the performance in this case. For this project, the "Wroom" module is used.

You can find a more in-depth specifications comparison between the "Wroom" and "Wrover-B" module online[15]

Figure 2: Example model of an esp-32 wroom dev kit[7]

The following specifications are for the ESP32 Wroom module. Many similarities can be seen to the Wrover-B module:

- 2x CPU Cores
- From 80MHz, up to 240MHz CPU clock frequency
- Three full- or half-duplex UART controllers, supporting common interfaces such as RS232, RS485 and RS422
- A sleep current less than 5 μA
- 150Mbps data rate
- 20dBm antenna output power
- 802.11 b/g/n, 2.4GHz network support.
- 4MB flash memory, with support for external memory
- Operation current at 80mA
- 38 pins, supporting a wide array of functions

Further details can be read in the official datasheet[16].

The second core comes in handy when transmitting data, as it handles all the WI-FI-communication, offloading a lot of work from the primary CPU. Even at 80MHz, it handles the workflow quite within bounds.

The Team would have liked to test the actual power consumption of the ESP32; however, lacked the equipment to do so. Still there are estimates in the datasheet that can shed some light on how much power the different power states consumes. [17]

Power mode	Description			Power consumption
Active (RF working)	Wi-Fi Tx packet			Please refer to Table 15 for details.
	Wi-Fi/BT Tx packet			
	Wi-Fi/BT Rx and listening			
Modem-sleep	The CPU is powered on.	240 MHz*	Dual-core chip(s)	30 mA ~ 68 mA
			Single-core chip(s)	N/A
		160 MHz*	Dual-core chip(s)	27 mA ~ 44 mA
			Single-core chip(s)	27 mA ~ 34 mA
		Normal speed: 80 MHz	Dual-core chip(s)	20 mA ~ 31 mA

Figure 3: Current draw estimates

Power mode	Description			Power consumption
			Single-core chip(s)	20 mA ~ 25 mA
Light-sleep	-			0.8 mA
Deep-sleep	The ULP co-processor is powered on.			150 μA
	ULP sensor-monitored pattern			100 μA @1% duty
	RTC timer + RTC memory			10 μA
Hibernation	RTC timer only			5 μA
Power off	CHIP_PU is set to low level, the chip is powered off.			1 μA

Figure 4: Current draw estimates

Although low power is not the highest priority of the ESP32, being more necessary in the CC1312R sensor node, it would be preferable to optimize the ESP32 to have a low-profile power draw, as to not waste unnecessary energy.

1.3.2 Modules in the Uplink

Throughout the project, multiple libraries are utilized in the ESP32-programming. This section will shortly describe each of them, giving some examples of functions used in each package. Firebase is the most notable one, making our Firebase communication possible.

1.3.2.1 <Arduino.h>

<Arduino.h> is an included, although not necessary package used in the code, as the team could use standard C functions to handle the same functions as used in this package. The <Arduino.h> is used to handle serial communication in an easy and fast way, compared to manually setting it up in C. The ESP32 has its own specialized software for easy low-level programming, called ESP-IDF, which the team abstained from using. There also exists a library that handles UART-communication specially for the ESP32 called “esp32-hal-uart”

which can be used but will not be covered in this text. In a future revision, the <Arduino.h> libraries is very likely to be removed, and its functions replaced by better optimized functions.

1.3.2.2 <stdio.h>, <stdlib.h>, <string.h>

<stdio.h>, <stdlib.h>, <string.h> are standard C libraries, frequently used in all kinds of code. The <stdio.h> library is not currently in use, but is under consideration as it has functions that easily helps replacing the existing function that are using <Arduino.h> in the current code. It handles I/O operations, such as reading serial input. <stdlib.h> has some conversion and memory allocation functions which are considered handy. <string.h> enables handling of strings in C, and this project mainly utilize strlen, strcpy, strcat and strtok. Strlen is a function that returns the length of an input string. strcpy is used to copy the string to a new variable. strcat appends a string to the end of another string and saves it. strtok breaks the string into multiple new strings, separated by a delimiter defined by the user.

1.3.2.3 <Wi-Fi.h>

<Wi-Fi.h> is the main module for handling WI-FI and its functions. It is utilized to enable communication through the integrated WI-FI-antenna. Some examples of functions used in our code:

Wi-Fi.begin(WI-FI_SSID, WI-FI_PASSWORD);

- Uses the input credentials set in WI-FI_SSID and WI-FI_PASSWORD to establish connection with the set SSID. The credentials need to be defined before being called by this function to work properly.

(Wi-Fi.status() != WL_CONNECTED):

- Checks the status of the WI-FI-connection. If it is flagged as false, it will attempt to reconnect until a success-flag is returned.

Serial.println(Wi-Fi.localIP());

- Prints out the local IP address received from the WI-FI-router when connection is established.

<Wi-Fi.h> is considered an Arduino module; however, it is independent of the main Arduino.h library and can easily be used on other platforms without dependencies.

1.3.2.4 <FirebaseESP32.h>

<FirebaseESP32.h> is a standard firebase library specialized for supporting the ESP32 modules. It is one of the hallmark libraries for our project, as it enables communication directly with the Google Firebase database. It has functions that handles compiling the input variables and addresses to JavaScript Object Notation format (JSON for short) and posting the JSON file to our firebase database. The library boasts quite intuitive functions, enabling the user to create, update, append, get, stream and more (as described in the official documentation), which helps to easily create a system that keeps the firebase database up to date.

The code is easy to handle, using if-loops to run the operations. As seen from this snippet of code

If (Firebase.setInt(firebaseData, Nodelight, light))

The operations always start with an “if”, or alternatively “while”. The statement inside the parentheses is considered a Boolean true if it can reach firebase, so an “if” will run once, and a “while” will run continuously as long as it keeps the connection to firebase alive.

“SetInt” defines the operation and the variable type. In this case, it means that it wants to write to the firebase, making a new node or overwriting a previous node. “Int” is of course short for “integer”, which is one of the datatypes supported by firebase. Other data types available are floats, Booleans, strings and JSON data packages.

“firebaseData” is the actual data file to be transferred, and has to be created earlier in the code, by using “FirebaseData DataName”. “Nodelight” is a string which is treated as the address here, containing the NodeID, followed by the name of the value to be transferred. This is created using the strtok and strcat functions mentioned earlier. “light” is the integer with the value to be transferred.

The Firebase library also has a handy function for reporting errors for the uplink;

firebaseData.errorReason()

firebaseData.errorReason()

This function has built-in predefined errors, which contains messages for the most common faults that can occur. By serial printing this, or sending it somewhere to be read, one can get feedback on why the database might not update properly. It does not have error messages for

every situation however, as the team have experienced situations where the errors were not in the pre-defined error-set, which requires manual troubleshooting.

The following actions for handling data, are available with the Firebase library:

Read data (GET)

- Receives the data from the address given to the firebasedata. Function. The data can then be saved, printed or modified to be re-sent back through other functions. Example:

if (firebase.getFloat (FirebaseData, "/Destination/ValueName")){do whatever}

Store data (SET)

- Stores the input data to the given firebase address. If the address already has data, it replaces it, otherwise it creates a new (folder) or subchild. Example:

if (firebase.setInt(FirebaseData, "/Destination/ValueName", IntValue)){Do stuff}

Append data (PUSH)

- Append attaches the data to the end of the previous stored data. This can be useful when making int/float arrays or writing strings. As an example, if the stored string in the Firebase database is "Hello", the function can add " World!" to make the new stored string read "Hello World!". Example:

if (firebase.pushString(FirebaseData, "/Destination/ValueName", String-to-add)

Patch data (updateNode)

- Patches certain nodes or subchildren data, without replacing all of the data in the node. For example, you can change the value of one subnode without rewriting the whole node. Example:

if (Firebase.updateNode(FirebaseData, "/Destination/ValueName", updateJSON){execute stuff}

Delete data (deleteNode)

- Deletes a node, including all its children Useful for removing clutter, values no longer in use, temporary values etc. This function does not need an if-statement.

- Only has one command:

Firebase.deleteNode(firebaseData, "/Destination");

Filtering, streaming, backup, and error handling functions are also available, and reading the attached or the official document is highly recommended if working with the firebase library, which can be found online[18].

1.4 Hardware

1.4.1 Capacitors and coils

A capacitor consists of two conductive surfaces separated by an insulating material (called the dielectricum), with the capacity to store an electrical charge on its plates [19]. The capacitor will charge when a voltage is applied, resisting changes in voltage once charged, and reversely discharge to an external circuit when the voltage sources is removed, acting as a temporary voltage and current source. This physical characteristic may be used for averaging voltage peaks and throughs of the AC component, and thereby reducing AC ripple from voltage sources[20]. There are both bipolar and unipolar capacitors, depending on the dielectricum used. Specifications encompass physical size maximum charge (in Farads), maximum working voltage (in Volts, at which the capacitor can reliably operate without concern about breakdown or changes in performance characteristics), leakage current, maximum and minimum temperature and durability to name some.[19] In order to reduce ripple, decoupling capacitors should be used at inputs, outputs and for every active component to average out peaks and throughs of the AC component.

The most common types of capacitors are ceramic, electrolytic and tantalum. Ceramic capacitors range from pF to μF and are small, one common application is decoupling of DC for ICs. Tantalum ranges from $100nF$ to several hundred μF. They are bigger than ceramic and smaller than other electrolyte capacitors. Compared to other electrolyte capacitors they are more accurate and has lesser production variance. Electrolyte capacitors such as aluminium

range from $100nF$ to several F and are typically larger in physical size than comparable Tantalum capacitors. The aluminium electrolytic capacitor has a shorter lifespan due to a quicker decomposition of its dielectricum, affecting its internal charge/discharge characteristics[19] however they are still widely popular as they can have large capacities and are relatively cheap.[20]

An inductor (coil) consists of numeral turns of wire around a ferromagnetic core or an air core[1]. The inductor will build a current flow when a voltage is applied with the energy stored in a magnetic field. The field will collapse if the applied voltage source is removed, and the stored energy will be returned in the form of a voltage spike.[20] Inductors may be used in conjunction with resistors and capacitors to filter unwanted frequencies from a signal and are often found in power supplies trying to counter current fluctuations. Inductive effects exist in various components and should not be ignored in embedded system design.

1.4.2 ESR and noise

Equivalent series resistance is the dissipative factor expected when using capacitors at various frequencies. As a rule of thumb, the lower ESR the better. Electrolyte capacitors will typically have much higher levels of ESR than for instance ceramic capacitors, a standard electrolytic 22 μF capacitor is expected to have between 5 and 30 Ω while a standard ceramic will have $10 - 100m\Omega$.[20] The leakage current of capacitors hints to the fact that even under DC conditions the resistance of the capacitor will not be infinite, and this resistance is presented as R_s . With AC the level of dissipation is presented as a function also affected by the magnitude of parallel resistance R_p R_d as stated in the following equation:

$$(1) \qquad ESR = R_s + \frac{1}{\omega^2 C^2 R_p} + \frac{1}{\omega C^2 R_d}$$

Equation 2, ESR

Stray Capacitance is capacitance that is most often not created "by design" but arises when two conducting surfaces are relatively close to each other.[19] This is a behaviour designer has to be aware of, both in printed circuit board (PCB) desig19n and IC design, especially at higher frequencies, where even low stray capacitance can be of importance for the functionality of the PCBs and ICs. Thermal noise is generated by thermal agitation of electrons inside conductors, the noise in a resistor is described by Nyquist and Johnsons theory:[21]

$$(2) \qquad U_{eff} = \sqrt{4kTRB_n}$$

Equation 3, Ueff

Where k is Boltzmann's constant, T is temperature (°K), R is the resistance and B_n is noise bandwidth. Since low ESR components produce less noise, they are preferred in noise sensitive applications.

1.4.3 Rectifiers

There are two important measures that defines the quality of the voltage supplied from a power supply; ripple voltage V_r and offset voltage V_{off}:

- Voltage Ripple V_r of a full rectifier is the unwanted peak-to-peak ripple voltage which may be viewed as a superimposed AC voltage upon the rectifiers designed DC voltage. V_r consists of an AC component with a given frequency f and peak-to-peak amplitude V_p.
- Offset voltage V_{off} is the magnitude of the DC voltage that when applied with appropriate polarity reduced the dc offset at the output to zero.[22]

A voltage regulator is a semiconductor device with the ability to convert a given DC input to a different DC output and are often used to supply a constant voltage. The switching regulator variant of voltage regulators can either step up (boost) or step down (buck) the supply voltage. Switching regulators are energy efficient and switches a power transistor (MOSFET) at their output. and offer voltage boost as a marked advantage over linear power regulators, however as a trade-off they produce more noise (primarily ripple voltage noise) than the linear power regulators due to their switching nature. The conversion process has an efficiency loss and the regulator itself uses a current (quiescent current). The efficiency of a converter is given by:

$$\eta = \frac{P_o}{P_i} \quad (3)$$

Equation 4, effeciency of a converter

The conversion loss is less in switching voltage regulators than in linear voltage regulators. There are various sources of noise and disturbance in a circuit. Electromagnetic interference (EMI) encompasses any external noise by any combination of electromagnetic effects. RFI is high frequency noise picked up in the power line (audible range and beyond) [1919]. Capacitive coupling is the coupling of electric fields and is caused by the signal on one wire capacitively inducing a phantom signal in an adjacent wire through its associated electric field [220].

Inductive coupling is the coupling of magnetic fields and is in the same manner a signal inducing a signal in an adjacent wire.

Printed-circuit boards are epoxy-bonded fiberglass sheets plated with copper. The interconnections of the circuit are formed by etching away parts of the copper plating leaving traces[20]. PCB boards may have multiple layers, whereas a multilayer setup aids in constructing of solutions that may reduce different types of noise and improve performance by optimizing e.g. trace length and thickness and avoiding unnecessary loops that may act as antenna. Traces should in general change direction by 45-degree turns unless merging with other tracks in a join where a 90-degree angle may be recommended.

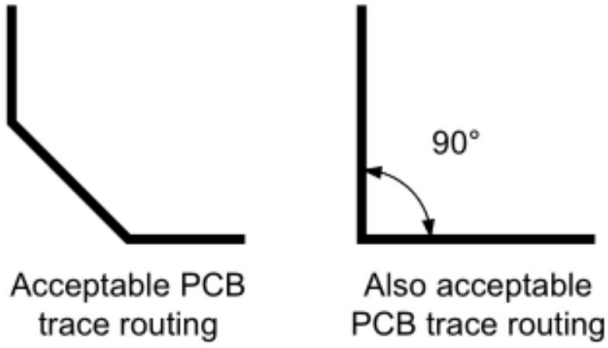


Figure 4. Acceptable PCB Trace Routing

PCB trace routing[23]

Also tracks should be fanned out to ease track routing. The EMI potential may be lessened by matching the power and ground traces with respect to location.

1.4.4 PCB trace design

The trace design and layout introduce several challenges and opportunities to circuit design. Signal traces should be as short as possible to prevent extraneous signals to couple into the signal path.[24] Particularly input terminals of analogue have higher impedance than the output or power supply pins.

For instance, you can build PCB resistors with your traces that span between components. You can build capacitors into the board with traces, soldering pads, and parallel traces. Inductors come from loop inductance, mutual inductance, and vias. Capacitance may be introduced by traces, soldering pads, and parallel traces. Changes in voltage with time ($^{\delta I}/_{\delta t}$) on one trace

could generate a current on a second trace. If the second trace is of high impedance, the e-field creates current, which converts to voltage. Traces placed in proximity may introduce capacitance defined by the following formula:

$$(4) \qquad C = e_0 e_r \frac{wL}{d} [pF]$$

Equation 5, capitance

where w is the thickness of the PCB, L is the length of the PCB, d is the distance between the two PCB traces, e_0 is the dielectric of air, e_f is the dielectric constant of substance coating relative to air.

Inductors may be introduced by loop inductance, mutual inductance and vias. Also caused by traces placed in proximity, either on top or side by side. Where changes in current with time ($\delta I / \delta t$) on one trace may generate a voltage in the same trace, and thereby initiate a proportional current in the other trace due to mutual inductance. These phenomena are in general more seen in environments with larger, seemingly instantaneous switching currents. The potential noise from EMI sources should be separated from quiet analogue lines.

Unmatched power and ground planes leave the design exposed to unwanted noise and may radiate noise off the board for induced voltages where total loop area will act as an antenna. An alternative solution is to introduce a ground and/or power plane. Ideally the ground plane should be uninterrupted to reduce the injected noise caused by $\delta I / \delta t$ which is change of current over time. The ground plane should be used as return path whenever possible. If possible, the digital ground plane should be separated from the analogue plane with a break, however breaks may also degrade the quality of the ground plane. Interruptions in the ground plane from signal traces should be vertical to reduce interference from the ground-current return paths. Placing the ground plane on the top of the board reduces EMI, however demo boards often use the top plane, alternatively a star connection strategy may be used should ground plane not be available.

To calculate the trace width the IPC-2152 method can be used, taking into consideration dimensioning maximum current, allowed temperature rise in path (combined with board substrate), copper thickness, nearest plane, board thickness, board substrate type, safety margin and so forth.[25] Most traces will be of constant width. However, it is important to keep in mind for ground connections that the thinnest width in a ground trace will be the effective

width from start to end. Ground loops are to be avoided and ground return paths should be as short as possible with minimal inductance. Digital currents should not pass analogue devices as they while switching are large for a moment, combined with the effective inductance and resistance. Please refer to the previous discussion of inductance between paths. A similar phenomenon is found with high frequency currents crossing lower frequency devices. The effects of the interference for the ground plane or trace inductance may be determined by the following formula:

$$V = L\frac{\delta i}{\delta t} \quad (5)$$

Equation 6, interference in traces

Pads are used to mount component pins and has various shapes (for instance round, rectangular or oval). An array of pads grouped to form a component package is known as a footprint and there are several pre-defined standards for footprints. Vias are plated holes used to connect tracks on different layers. Fills are used to shield sections of the PCB and to form circuit paths, in a Faraday cage like manner. They are most efficient if they are placed in ground layers on each side of the conducting layer(-s). Placing ground fills in and around analogue sections will isolate them from crosstalk.[20] In addition to several layers, PCBs can have printed layers known as silkscreen, this is mostly used for text and component outlines.

Inherently noisy and low noise parts of the PCB should be physically separated. For mixed domain layouts it is recommended to separate the analogue and digital components. Analogue components are more often exhibit less noise margins, while digital components in generate and tolerate more noise. It is recommended to place digital and higher frequency devices closer to the connector to prevent high frequency noise injecting into the lower frequencies' devices.

One area to be particularly cautious of is with the input terminals of analogue devices. These terminals normally have a higher impedance than the output or power supply pins. Traces between components may act as resistors. Separating quiet analogue lines from noisy I/O ports will help eliminate potential EMI sources. Implementing low impedance power and ground networks while reducing inductance in conductors for digital circuits and reducing capacitive coupling in analogue circuits is recommended.

1.4.5 Crystal oscillators

A crystal oscillator is an incredible essential component in any modern smart application, this component is the heart of the application; keeping everything in sync and enabling the MCU to accurately keep track of time. The way an crystal oscillator operates is by creating a feedback loop into an inverting amplifier, this part of the circuit is usually embedded into the MCU itself, then the remaining circuit is the crystal oscillator itself and two load capacitors which makes up a pi filter, this will make the circuit pulse with a fixed frequency.

In order to achieve this fixed frequency, the varying amount of capacitance found spread throughout the oscillation circuit must be accounted for. This is the duty of the load capacitors, if the capacitors are poorly matched or non-existing, then the frequency may not be exact, and this will cascade unto the MCU which may suffer from problems relating to the mismatch between the internal clock and real time operations. The load capacitors can be calculated by using this equation:[26]

$$CL = \frac{C_{x1} \cdot C_{x2}}{C_{x1} + C_{x2}} + C_{stray}$$

Equation 7, load capacitors

CL is the crystal load capacitance, Cx1 and Cx2 is the load capacitors and Cstray is the combined stray capacitance combined with the parasitic capacitance found in the terminals of the crystal oscillator. The load capacitors together with the stray capacitance and the capacitance found in the terminals of the oscillator will make up the total load capacitance in the crystal oscillation circuit. Usually the value of the stray capacitance and the terminal capacitance found in the terminals of the crystal oscillator can be assumed to be within the range of 2 – 8 pF.

ESR is a parameter found in most electronic components also the crystal oscillator, here it needs to fulfil a requirement for the crystal oscillator circuit to function properly. This requirement is defined by the following formula:

$$ESR < \frac{R_{ned}}{5}$$

Equation 8, ESR to Rned

If this requirement is not fulfilled, then the oscillator circuit will not function.

1.4.6 Decibel sensor

The decibel sensor's purpose is to generate a voltage spike that can be used to gauge the current noise level. This is done by first filter out unwanted high frequency noise from the microphone, then amplifying the resulting signal through an operation amp. The amplifying should be adjustable such that an optimal amplification is achieved, then the signal is ready to be measured by an ADC.

2. Method

The method followed the breakdown structure, as visualized in Figure 5.

Figure 5: Work breakdown structure

2.1 System link

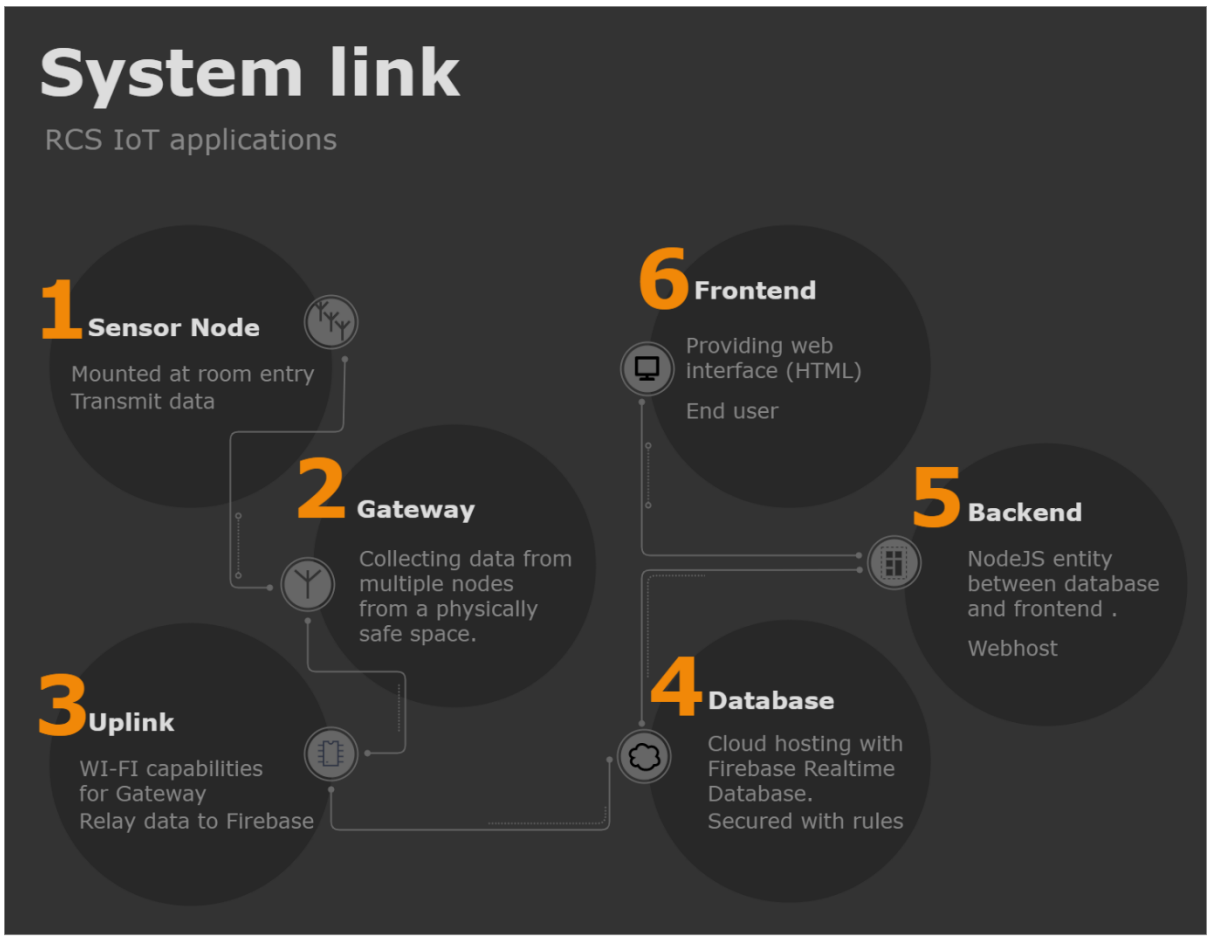


Figure 6: Full system architecture

The system link consists of different modules:

- The node sampling and recording data on the room/door state. Using sub 1 GHz communication with the gateway and by using the sensor controller capabilities within the CC1312R chip MCU of the node, recordings of the room/door state can be transmitted to the gateway with the node using as little power as possible. In a network, multiple of these hardware modules are needed if several rooms/doors need to be recorded.
- The gateway collecting the nodes recording. Also known as the master node or the collector, the gateway communicates with all the nodes in the system, collecting the nodes recordings and relaying the information to the uplink. This piece of hardware is needed as the nodes communicate using sub 1 GHz. As an example, the CC1312R MCU on the gateway takes the recording that the node sent and with the use of UART sends this recording to the uplink.

- The uplink. Connecting the gateway and the database of the system, as the gateway does not itself have 2.4 GHz capabilities. This hardware is giving the gateway Wi-Fi capabilities and will upload the recordings from the gateway to firebase, in addition to processing the string received from the gateway. In this thesis, an ESP32 is used as the uplink MCU.
- The database. Providing an online NoSQL storage space for the recorded data, secured with security rules. The database of choice is Firebase Realtime Database.
- The backend. Designed to serve a real time connection between the database and the frontend. The backend must also feature data processing capabilities and a modular design. The backend consists of Node.js entities which interfaces with the front end.
- The frontend. Featuring the web browser interfaces with the backend and should be accessible in a web browser. The web page was marked-up and showcase real time data for each specific Node ID, meaning the different nodes.

2.2 Work Packages

To achieve a low power IoT sensor system the project was divided into packages that would be worked on such as:

- Designing the Node and the Gateway with Wi-Fi capabilities. The team needed to design a PCB for a CC1312R-chip working as our main node. This process includes adjusting resistors, capacitors, and inductors, constructing the right filters, inputs/outputs and maximizing the efficiency of our system. Choosing and designing the right type of antenna is also a major part, along with constructing a case for the node. Having a single chip Gateway meant that the gateway got Wi-Fi capabilities, along with modularity.

- Connecting the Node to a Master-Node/Gateway. The node itself should not directly communicate with the server, due to security concerns. The team solves this issue by offloading most of the security handling to a Middleman, referred to as Master-Node/Gateway.

- Connecting Master-Node to our server. The master node will relay the data to a web server through a secure connection. The web server will read the transferred data and save, process, and forward the relevant information to a real time database system.

- Connecting server to real time database. The real time database will read and write the relayed data for long- and short-term storage. The system will thus be isolated from the master-node as well as the web page and App and act only as a designated storage space. The information will be categorised and formatted for easy shareability.

- Constructing and connecting a web-based server. To handle the transferred data, The group one mustneeds to set up a Firebase database, which will store and send all the necessary data. The team also wishes to use a Node.js as an extra security measure for our system, as security is a huge consideration throughout the whole project. A Node.js server will reduce any potential damage from any break-ins.

- Designing a GUI for the website. To easier present the data to any viable user, the simplest solution is to set up a website which can access the stored information through either a web browser, or potentially through a mobile app. The website will be constructed using CSS and HTML, and Python will be used for any potential app.

- Designing security measures. Proper security is needed in each module, to secure the data from any break-ins in our system. The data could be used in a critical system, where unwanted access to the information could give unwanted consequences such as break-ins, changes in the system code, or other exploitable activities.

- Testing and debugging**.** A crucial part of the project during development is testing out the system, tweaking it to maximize efficiency and reducing bugs/noise and other unwanted elements. Multiple bugs and nuisances are expected from Node MK1, which will hopefully be resolved by later versions, MK2 or MK3.

2.3 Node and the Gateway

The research on the system started early January, and there were multiple consultations with the professional supervisor and the scientific counsellor. The team also met with TI-employees the 14. February at the electronics and technology day at NTNU campus Gløshaugen, Trondheim. Regarding the research on firmware and hardware specifically, the team members used internet resources like google when researching, along with textbooks.

Researching code composer studio used for making the firmware for the node and the gateway, Texas Instruments resource explorer[27] came in handy as documentation on the different api-s, the CC13x2 and CC26x2 SDK, and example projects helped when making decisions.

2.3.1 Purpose

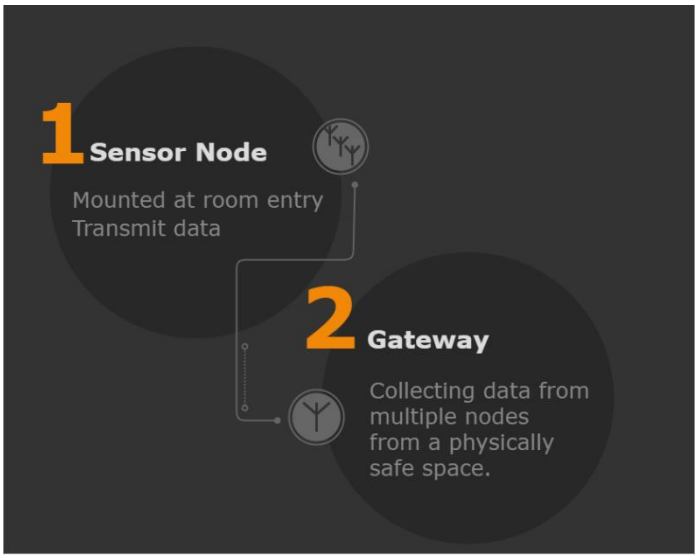


Figure 7: Network architecture

The node had criteria's that needed to be fulfilled. Low power consumption and good signal integrity meant the part of the system most physically exposed would need zero maintenance for a substantial amount of time and have a robust connection to the gateway. The gateway needed to communicate and collect data from multiple nodes. See Figure 7 for the node and gateway network architecture.

2.3.2 Sub 1 GHz

The team decided to use the 863 to 870 MHZ band as it was within the license free operation domain. The CC1312R chip is capable of different types of radio specification. In this project the team will however use 2-gfsk (Gaussian frequency shift keying) modulation with 50kbps, a frequency of 868Mhz and a bandwidth of 200 KHz. The requirements for this frequency are 1% duty cycle or LBT with AFA. Maximum ERP is also 25mW for this 868Mhz frequency. The modulation was decided upon as the 2-gfsk is the default modulation for programming and 50 kbps gives the team room for sending a lot of data.
When it comes to network topology the sub 1 GHz technology enables low power and supports sleepy end nodes - battery operated devices that are in their lowest power state most of the time and wake up based on external triggers or timers to transmit and receive messages.

2.3.3 Programming the Node and the Gateway

2.3.3.1 TI-RTOS

Is an embedded tools ecosystem for use inside CCS and a wide range of Texas instrument processors. It includes a real time operating system component called 'TI-RTOS kernel' along with additional components that support device drivers, power management, file systems, instrumentation, networking connectivity stacks and inter processor communications. TI-RTOS was chosen to be included in the programming of the CC1312R chips as it gave modularity to the project along with the fact that most demo projects and Literature from Texas Instrument concerning The CC1312R Used TI-RTOS. The Kernel gave several capabilities to the programming project, like tasks and events, where tasks are in short functions executed in terms of priority level. This as an example would be two tasks that sample ADC's, where one has priority 4 and one has priority 3. The priority 3 would run first as the bios is started and then the one of priority 4. Events are more like configuration of tasks, where the bios makes the processor sleep until the event is triggered by something and the task can run. Certain initializations and prerequisites must be fulfilled if one wants to use these functionalities in a CCS project. By using the RTOS, the CPU can choose when it wants to sleep to gain ultra-low power consumption on the node.

2.3.3.2 Sensor-Controller Studio

The integrated development environment Sensor Controller Studio is the preferred client for programming the Sensor Controller within the CC1312R chip and is independent of the CC1312R's Processor. The IDE has different functionalities and does not per se program the sensor controller, but instead it makes drivers for coding in CCS. The Sensor Controller Studio Language looks like the C language but has instead its own APIs for almost everything, which means new developers need to rely heavily on the Sensor Controller Studio Help book, example projects and online forums. The Sensors to be implemented would use as low power as possible, this means that the sensor controller would use a 2 MHz clock. Sensors dependent on the ADC would use 12-bit as it was the default for the CC1312R sensor controller.

2.3.4 Firmware architecture

Using the selected IDE for programming the CC1312R MCU on the node and the gateway. Two different projects were made for the custom node PCB that was planned to be made during the project. The projects made in CCS consists of an application running on top of TI's real-time operating system (TI-RTOS) see Figure 8. TI-RTOS includes the SYS/BIOS kernel, optimized power management, and peripheral drivers. The Easylink abstraction layer that sits on top of the RF-core driver runs alongside TI-RTOS and handles all radio network operations. In short, concerning the wakeonreednodefinal[28] and wakonreedgatewayfinal[29] projects, the application services interrupts from the reed switch sensor and processes commands received from the network. Choosing this architecture for our firmware was a decision that was made consulting TI-employees. Using TI-RTOS gives modularity to the project so it can be easily expanded later, Easylink was chosen as the team did not have much experience with programming the rf core driver.

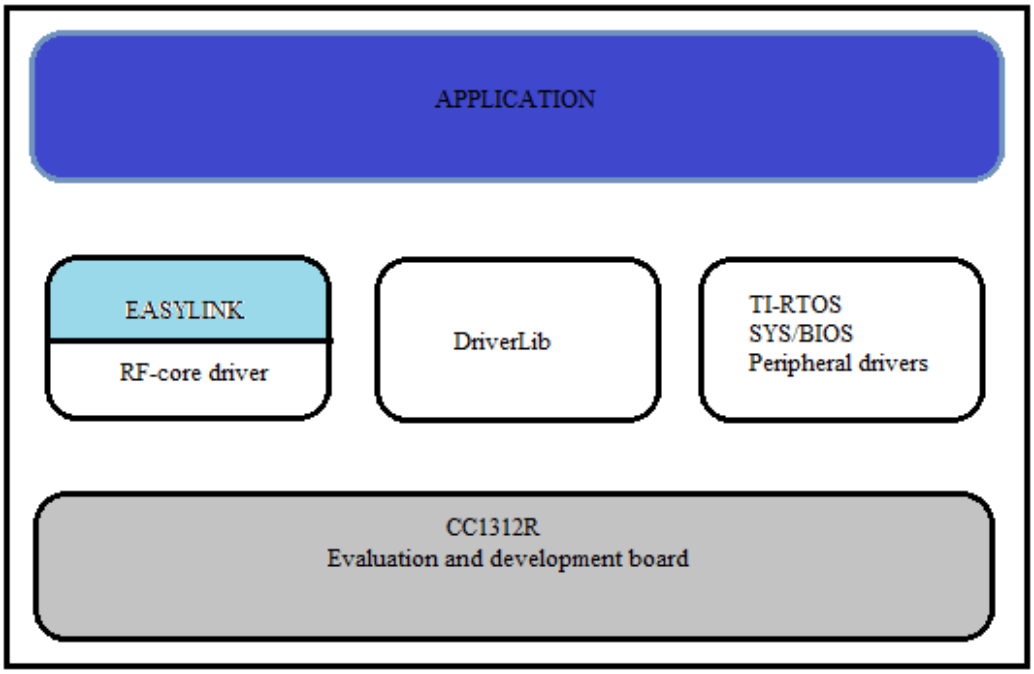


Figure 8: Project architecture

2.3.5 Firmware behaviour

The sensor firmware connects to the network and processes interrupts from the reed switch sensor. The firmware will keep the CC1312R in standby mode until an interrupt happens on

the reed switch sensor, that is the reed switch changes states. When the CC1312R is woken up by the reed switch sensor interrupt, it will assemble and send a packet to the gateway. Indicating the status of the reed switch sensor output. An unsigned integer 1 in the packet sent corresponds to no magnetic field present and a 0 indicates a magnetic field is present. The CC1312R will return to standby mode after either an ack-packet from the gateway is received. If no packet is received the node will send the packet 2 more times before giving up and returning to standby mode where the sensor firmware waits for a toggling of the reed switch. The packet sent from the node includes a node address, battery power, internal temperature of the CC1312R and internal time for the gateway application.

The Gateway waits in RX active mode waiting for a packet. When the packet is received the gateway checks if it is a valid packet before sending an ack-packet to the node. The gateway then unpacks the packet payload and extracts information. The data is then sent over UART to the ESP32 connected to the gateway on Digital output pin 1 and pin 16 on the esp-32.

Figure 9: Node behavior

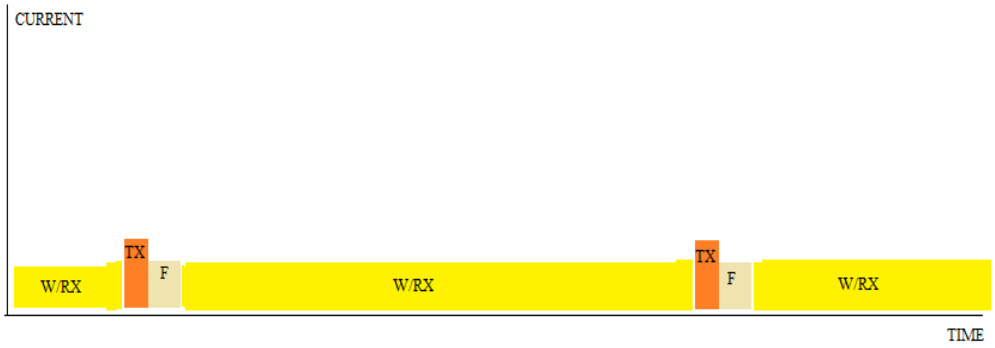
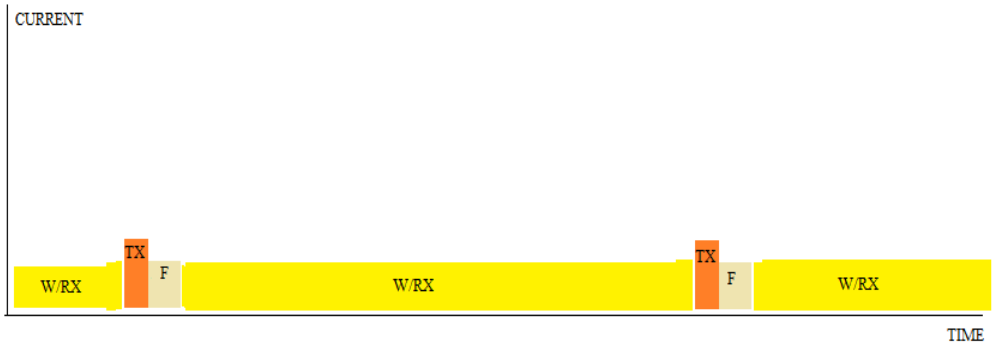


Figure 10: Gateway behavior



shows how the gateway waits in receive mode(W/RX), once it gets a packet it transmits an ack(TX), finishes whatever it needs to do with the data like relaying it to the esp-32 over UART(F), and goes into wait in receive mode again. This is the ideal behaviour for the node. The node can send up to 2 more packets if there is no ack- packet from the gateway, the node will then sleep until it is woken by the sensor controller alert. With this behaviour the firmware establishes a simple two-way communication, making it not only modular, but the parts of the system using batteries with limited capacity, low power.

There was also an attempt at sending multiple sensor data from the node and gateway on a 2-minute timer or whenever the reed switch switched states, this is the unfinished micreedlightnodefinal and micreedlightgatewayfinal projects. Although the behaviour of the node is almost the same as in the wakeonreednodefinal, excluding the 2 minute timer as then the timer would wake up the main CPU every two minutes and a packet would be sent, a bug in the code either on the node or the gateway would result in some of the sensor data to be displayed as 0 on the gateway. Having the ability to sample and send multiple sensor data from a node would give the system more parameters over surveillance of the room. Brightness sensors as an example would give the user of the system the ability to see if there was any light in the room.

2.3.5.1 Data flow

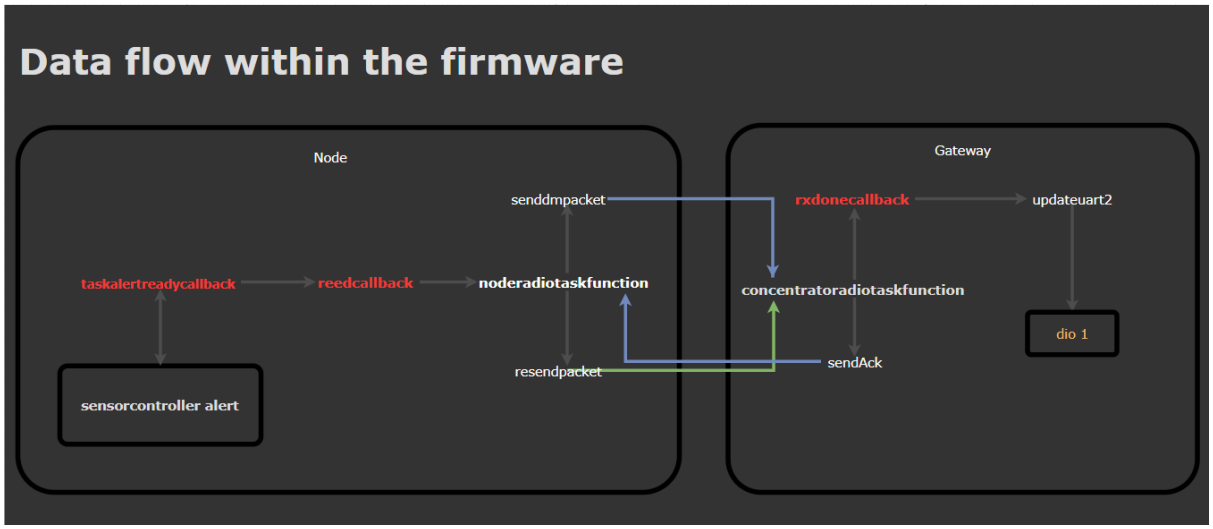


Figure 11: Data flow within firmware

Figure 11 is a representation on the important functions and callbacks in the 2 projects code which data passes through. All starts with necessary initilazations such as the sensor controller and task's in the firmware before the bios is started on either device. within the sensor-controller firmware execution code, if the sensor-controller firmware senses a change in state of the reed switch, the code sends an alert to the main CPU. The taskalertreadycallback handles this alert, signalling the sensor controller that the CPU is now awake. The taskalertreadycallback callback, gets the sensorcontroller output struct containing the reed switch state. And passes the value of the reed switch to reedcallback. This callback is used to transport the value to a noderadiotaskfunction, where the senddmpacket function puts the value into the payload of the packet to be sent. The packet is then transmitted over the radio, and the node waits in receive mode until an ack packet is received or a timer of 10ms runs out. If the timer runs out the packet is resent with the resendpacket function, the packet can be sent a maximum of 3 times during a CPU wakeup. When an ack packet or the final timer runs out, the CPU goes into sleep, waiting for another wakeup from the sensor-controller. The gateway, which is waiting in receive mode, when a packet is received checks if the packet is valid or not. If not, no ack packet is sent. If the packet is valid, the gateway unpacks the payload struct in the rxdonecallback and puts the different values in a helper struct. The helper struct is used when formatting the string that is to be sent over UART to the Esp-32. The updateurat2 function is executed and values from the helper struct is taken and formatted with the use of the sprintf function before it is sent through dio1 on the board to the ESP-32/uplink.

From the sensor-controller output structure, the data being, an unsigned integer gets transported and put in a struct before converted to a char in the payload struct. In the gateway the data is converted back to an unsigned integer residing in a struct where the updatuart2 function copy the value and adds it to a string as a a char. The string is sent and ends up on the ESP-32 where the data is formatted.

2.4 Uplink

When the team started discussing the idea for the uplink, the focus was set on a low-power system and a lightweight and easy-to-understand code. C was used as the primary language, due to the efficiency and simplicity if it, yet being quite powerful in the right hands. The team also has prior experience handling C, and along with its high modularity and compatibility made it the obvious choice for the uplink, although Python was considered, and the programming could easily be reproduced in Python if one rather wants to use it or any similar MCU-supported languages.

2.4.1 Purpose

The ESP32 exists in the system to complete the connectivity from the gateway to the database. It accepts the data sent from the gateway, processes it to the standard Firebase file format, and handles the sending to the database via 2,4GHz Wi-Fi connection. This helps offloading the workload from the gateway, contributing to a quicker and more stable system, and allows communication between the gateway and the database.

2.4.2 Drafting and prototyping

In the early stages of the project, the team utilized a Raspberry Pi 3B+ as a CC1312/ESP32 simulation to quickly produce a working demo as a proof-of-concept. This demo generated and sent a set of random data with a given NodeID. This was done to show the viability of the system, and to examine if the idea was achievable with the firebase ecosystem. The demo produced promising results, which strengthened the team’s reason to use Firebase as the standard database. After further testing, the Raspberry Pi was switched to the less powerful ESP32.

The Raspberry Pi 3B+ boasts a 1,4GHz Cortex-A53 processor, 1GB ram, and multiple other features fitting for a more advanced project. the team decided that the Raspberry Pi had too much overhead for the system, which meant a higher power consumption and a higher production cost, and costing about 35 GBP (~430NOK) compared to the ESP32 at 3 USD (30 NOK)[30] for a single chip and 10USD (~100NOK)[31] for a dev kit. The sizes also had to be considered, and as the ESP32 is notably smaller than the Raspberry Pi 3B+, and it was concluded to be better to scale down to the ESP32. This way, it could still achieve the same in-system performance, but at a much lower cost.

One of the differences with the Raspberry Pi is its Operating System-tailored hardware. This enables a higher-level interface, which for example enables low-level executions through command prompt allowing more ways to handle how one might program the ESP32-module. This is however an unnecessary component for the team, because of lack of experience using it compared to using a regular IDE, and such makes the setup more complex than needed, having other prerequisites than the chosen IDE Visual Studio Code.

The team have yet to properly test the large-scale performance of the system. The current code can theoretically allow for up to 255 sub-nodes per CC1312/ESP32 couple, and it could occur that it becomes too taxing for the ESP32, which could require consideration of a more powerful alternative such as the Raspberry Pi 3B+. It was however concluded that reaching this scale was unrealistic, as the system will never reach this capacity in any of the imagined set-ups and conclude that the ESP32 is quite capable considering the envisioned system size. If it should occur that 255+ nodes are needed, the team will rather solve it by spreading the workload over multiple ESP32, rather than collecting all the nodes on a single Raspberry Pi, considering the signal integrity throughout such a large system.

2.4.3 ESP32 Interfacing

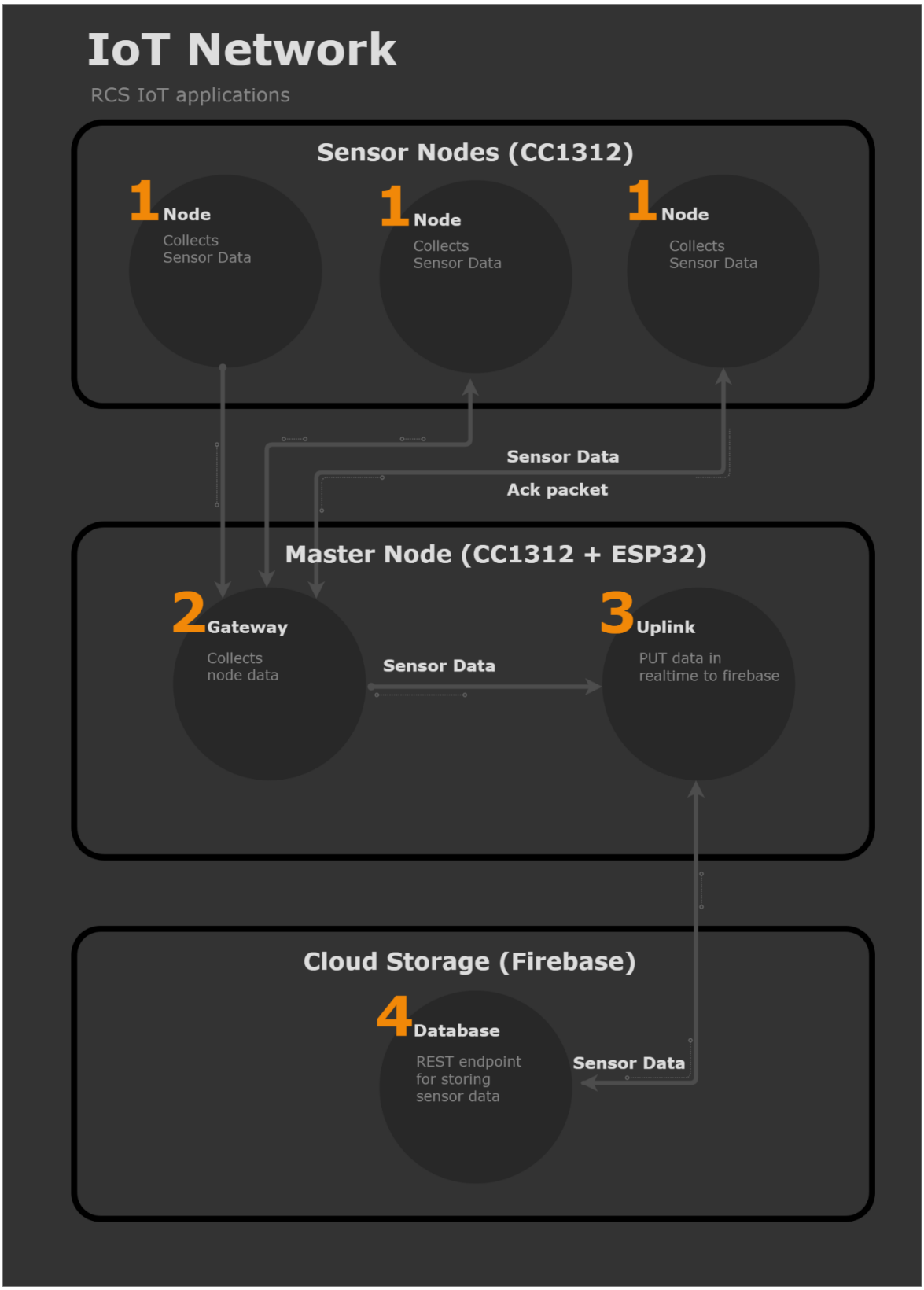


Figure 12: Overview of the interfacing of the Uplink

The uplink communicates with the gateway through an UART link using simplex communication. The team chose UART because of the simplicity it provides with only one pin at each end needed for the simplex communication, and it is easy to enable half duplex if needed, or even full duplex by using the corresponding RX/TX pair at the gateway and the node. UART is also preferable due to the asynchronous mode, which eliminates the need of a shared or external clock for the process. UART also gives an easy access to the data through a computer by using converter cable, where the user can read the input and output via serial.

The expected data is given by a standard format made and agreed upon by the team. The Uplink will expect a string in the format NodeID:Value1:Value2:Value3: where the NodeID has a length of 13 characters, matching an example ID "ntnuGateway08". The length can of course be adjuster as-per system, but this length was chosen to make an easily identifiable ID which supported multiple nodes. In this case, the "ntnuGateway" will be a common start-ID, signaling that the node is under the "ntnuGateway" system, while the number tells which node it is in the system, supporting up to 100 separate nodes. The values after the NodeID: are of any integer of size up to 16 bits, but preferably between the integer-range 0-100 (which an 8-bits integer covers) as to not flood the system with unnecessarily large values, which bloats the memory of the uplink. These represent the sensor data from each sensor in the CC1312-node. It currently reads three values, as represented in the string.

This format was chosen to always easily have an overview, enabling a quick identification of each value. This also shortens the troubleshooting procedure, as one can see where or which value might cause problems. This proved valuable throughout the project, as some errors occurred, and the formatting made the troubleshooting simpler.

After the data has been processed by the uplink, it has been converted to the JSON format and is sent via Wi-Fi through pre-built functions from the Firebase library. These functions neatly package the values into a JSON file and sends it to the Firebase database. JSON is the standard accepted format in the database, as it does not accept the standard string received by the uplink.

2.4.4 Operations of the Uplink

The code in the uplink is an attempt to achieve these steps, by using slick and intuitive functions. The setup starts with some standard procedures setting up variables and #defines, enabling and searching with Wi-Fi, and connecting to the correct firebase database. In this part, one will find common functions and operations for setting up the features of the included libraries, and as such will not be explained as this is done in the respective libraries must-read documentation. It is however important to mention the #Define FIREBASE_HOST and #Define FIREBASE_AUTH which defines which database the function will connect to, using the corresponding authorization key. This must be done in the code for each uplink that wishes to use another database, else all the data will be collected by the same database, which might not be desirable.

The main loop consists of self-made functions, each with its own operation-set, which also links into the next function in a chain of operations.
The code is started with checking the UART for any available data. If no data is available, it keeps searching until the "data available" flag is set. This way, the code will not waste CPU cycles doing the operations on no string at all. When data is available, it sets a Boolean true, which enables further operations on the string.

When the "Data available" Boolean is read, the next function initializes. The ESP32 then starts collecting the incoming string, saving it as a variable. The read data is also printed to serial to allow the user to check the incoming string. This way, one can see if any errors occurs during the transfer. At the end, it sets a "Data read" Boolean, enabling the next step.

The string is then split up to the correct parts. It first extracts the NodeID and checks the length of it. This way, one can ensure that the incoming string is not faulty. If the NodeID is too long or too short, it is considered unfit, and the function trashes it, and skips all following functions, starting the main loop over again. If the NodeID is the correct length, it continues operations as usual. This was done as a work-around of an error that reoccurred during designing and testing of the code. Multiple times, the strings ended up garbled, but due to no restrictions in the code to ignore the garbled string, it completed all the operations, and sent to the firebase database, which ended up with a lot of spam-nodes, as every garbled string was

unique. With the work-around, only the correctly formatted strings are allowed, which has worked neatly during all further testing, although it is in no way a stable long-term solution.

When the NodeID is the correct size, the code starts splitting up the rest of the string, utilizing multiple <string.h> library functions, which is a standard C-library with easy and intuitive functions. The values are collected from between pre-set delimiters, which in our case is the ":"-sign, as seen in the systems pre-defined format. The collected values are then converted to integers, as they are considered char-type when separated from the string, as to ensure proper transmission and readability to the database. The team decided to add serial printing of these values too, giving feedback about the resulting values. The following sketch show how the string is split into the multiple parts (Note that this figure show how it would split a string with 5 values):

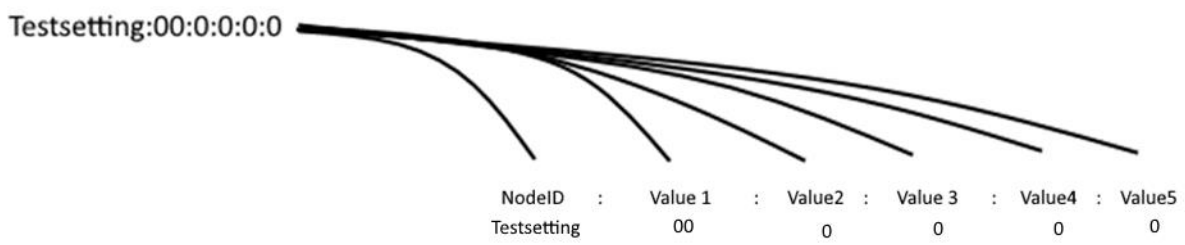


Figure 13: Visualization of how the string is splitted

The Firebase functions needs an address to send the data to. To solve this, the code appends each value-name from a pre-defined set, such as "temp", "doorstate" and "batteryPower", to the end of the NodeID, saving each one as a unique variable. These variables are needed in the next step when the firebase functions are called. When all operations within the function are done, it reports a new Boolean true to the next function to tell that the dataset has finished parsing.

The string is now properly processed and is sent to the database. The function
if (Firebase.setInt(firebaseData, "Destination", "Value"))
is called, which handles the conversion of the value into JSON format, using the address previously mentioned as the destination for the information. When everything is neatly packed into the JSON file, the data is sent to the firebase database defined in the setup of the code.

For every value sent, a corresponding error-report function is included, which prints the error that might have occurred if the function should fail by printing it to serial for easy troubleshooting. This way it is quicker to locate and solve any problems that have occurred when trying to send the data. The error-function has pre-defined messages for the most common problems that occur when dealing with firebase-sets, however it could occur that a problem is not defined, and as such it reports a blank error-message.

The code ends by setting all Booleans back to false and starting from the top again. Below, a simple function-flow can be seen, giving a clear overview of how the code operates at a higher level.

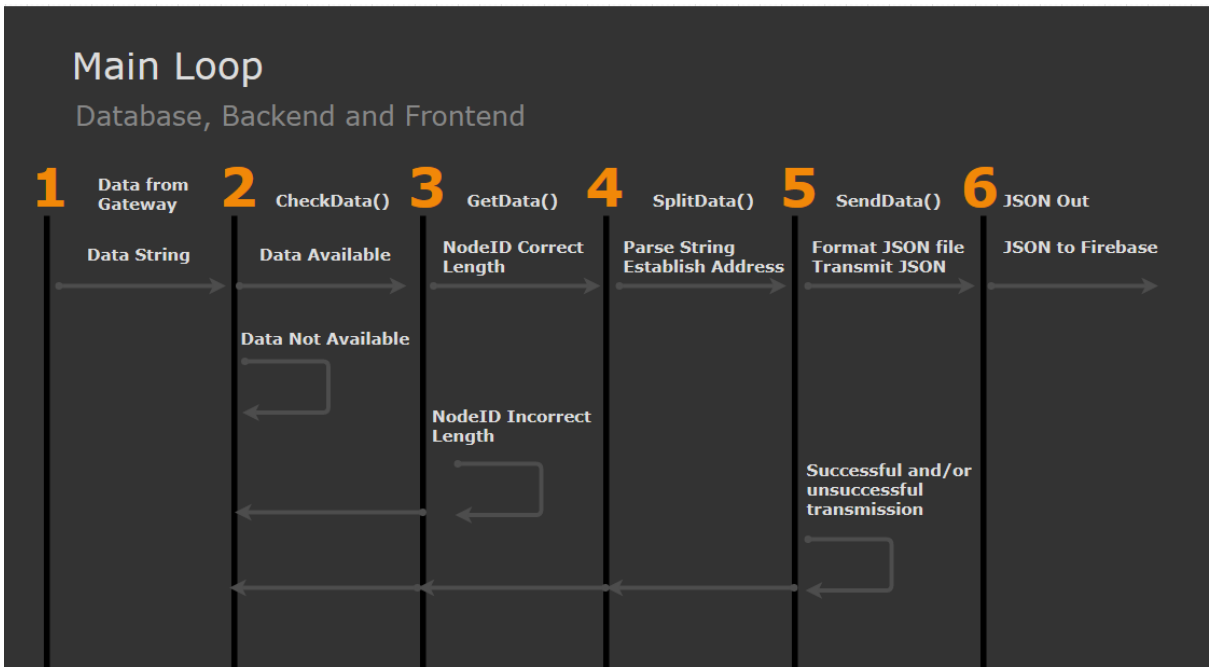


Figure 14: Function flow within the code

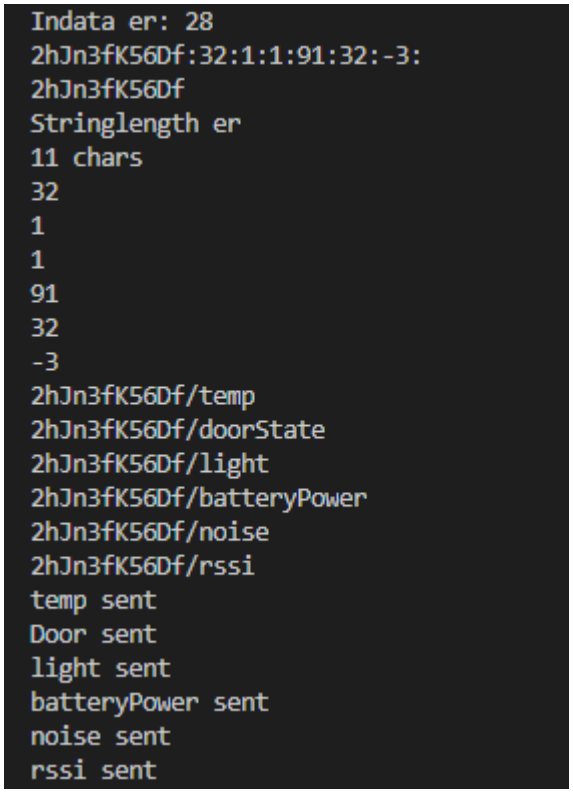


Figure 15: Serial output

This figure shows what the serial output to a computer reads during a successful transmission. Note that this includes values for sensors not included in the current version of the attached code.
It displays the length of the total read data, followed by the total string for verification purposes. It then extracts and checks the length of the string (11 chars were in this test, although the finished system uses 13 chars), and prints all the separated values, followed by all the addresses after appending them, and sends success-messages for each value that is transmitted to the firebase database.

2.4.5 Libraries and their usage.

Throughout the code, the project uses multiple libraries to enable certain functions to operate properly. This section will explore each function, making it clear how the libraries are use, and what lies behind the design of the uplink.

The CheckData() function uses Serial2.available() from the <Arduino.h> core to check if the DataAvailable flag is set at the UART2 channel. This function is also used in the start of GetData() to ensure that nothing has happened to the string in the meantime, although it might seem redundant for some.

GetData() also uses Serial2.readBytesUntil() to read the string until a defined char inside the parenthesis is reached, which in the case of this project is a NULL-terminator, which collects the whole string. The <string.h> library is then used to save the string using strcpy() and check the size of it with strlen().

SplitData() mainly utilize the <string.h> and the <stdlib.h> libraries. Strtok is used to collect each part of the string divided by the pre-set delimiter, saving it as a temporary variable. After ensuring the correct NodeID length, it converts the following values to integers by using atoi() and saves them as unique variables. This is a simple, yet highly effective way to parse the string as it handles the string directly, instead of operating with multiple temporary variables switching back and forth to correctly split each part.
Following this, strcat() is also used, which appends the saved NodeID with the destinations preset with for example "/temp", which is the destination mentioned earlier.

The last part which consists of converting and sending the data is easily done with the pre-defined function Firebase.setInt from the <FirebaseESP32.h> library. These are described more in-depth in "1.3.2.4 <FirebaseESP32.h>"

Throughout all parts, Serial.println() from the <Arduino.h> library is periodically used to send the string through UART, possibly read by a computer to ensure troubleshooting opportunities as mentioned in "2.4.4 Operations of the Uplink".

2.4.6 Programming software

The goal for our programming is to be light-weight, easy to understand and use, yet highly effective for our task. To achieve this, the team use simple pre-defined functions, and try to keep the rest of our code at C-level, knowing that it is highly efficient.

For the ESP32-module, The team used Arduino IDE at first, however due to much incompatibility, errors and Arduino IDE's general unwillingness and own defined strings known as string objects, it proved better to change the programming IDE. Microsoft Visual Studio Code was the best alternative after some consideration and replaced Arduino IDE. Visual Studio Code is a well-established and easy-to-use editor which supports a wide array of plugins in addition to being fast, which made the choice obvious for the team. One of the plugins used throughout the uplink programming is PlatformIO. This plugin has certain features that better support the ESP32, such as its own config file to easily set baud rate, CPU speed and similar hardware options, in addition to better integrating MCU interactivity than VSC already provided, giving more control of configuration and coding.

In the early phases, the team had to consider which programming language would be best to use for the uplink. two languages were considered; Python or C. As Python is an easy and fast-to-prototype language, the team tried to use it to quickly make a demo on the Raspberry Pi. However, as the team's knowledge of Python at the time was limited, it was decided to port it over to a C-prototype, which was used to further demo the code on the Raspberry Pi. After settling with C, as it is very lightweight and relatively fast, the team was able to more reliably write functioning code and could easily switch out the Raspberry Pi for the ESP32, without learning the additional Python-extension MicroPython, which optimizes Python code to better run on MCUs. Another advantage is the fact that C is common in the embedded systems-field, which makes it more compatible with peripheral systems. Troubleshooting can also be considered somewhat easier, as one can thinks of it more from a hardware-perspective, which helps if one understands how a computer "thinks". The advantage with Python is that it's great for demoing concepts requiring little code to prototype quickly, which is why the team considered it when building the first tests.

2.4.7 Test bench

If you lack the CC1312, or want to use another system, the ESP32 is fully compatible with any device that has UART. The only requirement thereof is that the string format is correct. During the middle phase of the project, due to the problems the Covid-19 virus created, the team had to make a CC1312-emulator to communicate with the ESP32 to check if the code was working as intended. This was solved by using a second ESP32 (any MCU with UART works fine too), wrote a simple random NodeID selector and value generator, which sent the data at random intervals. Next, connection was established from the CC1312-emulator's UART output to the regular ESP32's UART input. Note that the picture shows two cables, but only one is truly needed: CC1312 TX -> ESP32's pin 16 (or RX-pin if serial feedback is not needed). With the communication link up, the CC1312-emulator sends strings at random intervals, and the ESP32 should correctly pass it to the Firebase database, which means that the corresponding data can be seen in the Google Firebase webpage.

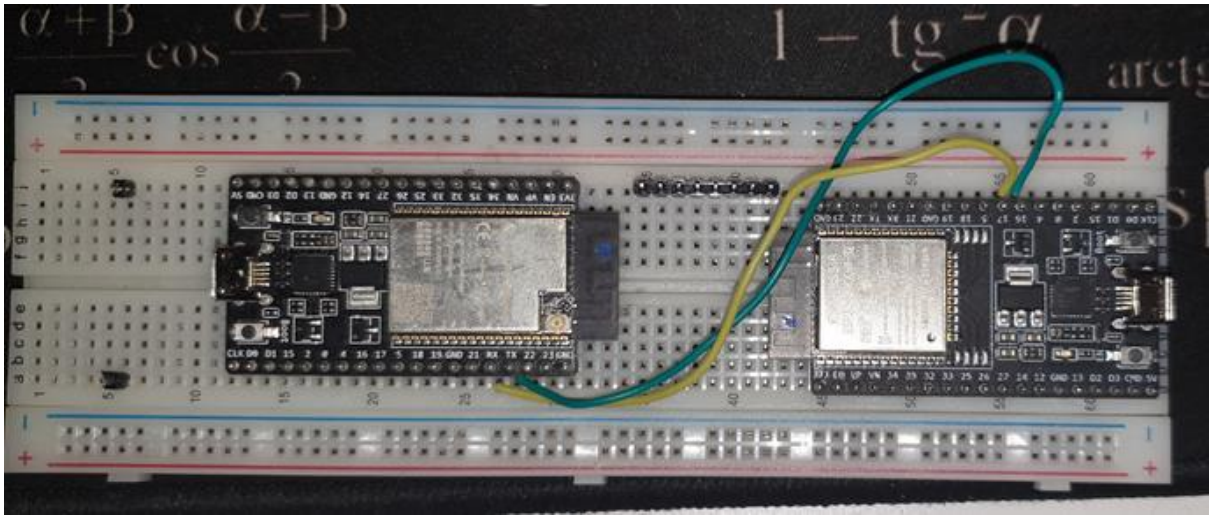


Figure 16: The test setup when programming the ESP32.

2.5 Database

2.5.1 Purpose

The database used in this thesis was used to store sensor data and had an easy and secure solution for updating and downloading this data. It was also a real time cloud storage platform which was easily accessible worldwide. It had a low downtime and offered safe access to requests and responses from other modules. Furthermore, it interfered with the uplink and used a shared data transformation protocol. The database featured security features which was to be used in conjunction with further security layers in the backend. Lastly the database was selected due to having an easy way of formatting the hierarchical data, to further utilize the data in the backend and user-friendly cloud interface.

Figure 17: Database

The database included a REST endpoint that may be accessed through the HTTP(S) protocol by computers and simpler devices such as microcontrollers. For applications with less data processing requirements, connecting, and interacting with firebase as a REST endpoint may be advised. It was a lightweight and simple way of interacting with the database that does not require any further library than support for HTTPS requests and responses. Thus, not requiring languages such as JavaScript and a runtime environment (i.e. Node.js). This duality allowed both simple uplinks and more complex applications such as Node.js to both interact with firebase unhindered, was a key point in the choice of database.

2.5.2 Data structure

The data structure in firebase followed a hierarchical tree configuration with JSON format with upper camel case. This data structure was configured as per best practice recommended by Google. [32]

Any data written to the database using Firebase-APIs would automatically follow the hierarchical structure as indicated by the client, if the client allowed the request. When using REST endpoint and the command PUT, it would either:

- Replace data in the database
- Create new data structure to the submitted URL

The system was not designed to save old values in the database, meaning overwriting old values. The requirements for the data structure was thus greatly simplified as PUT (i.e. setInt()) is used on the uplink, as explained in section 2.4.4 Operations of the Uplink. Old values would thus be automatically erased (as implied by the PUT), and new values was instantly available in their place. To summarize, the interfaces towards the database ensured short delays in real time updating of the database, a relatively simple data structure and a cost-efficient database plan (with low cost on data storage).

The JSON format enforces a standard dictionary with a standard Key-Value Pair. These keys and values are manually set by the client and faithfully replicated in the database. Nesting of data was avoided as per best practice by Google, and therefore the magnitude of children was kept to a minimum. This is a definitive advantage as Firebase-APIs are set up to retrieve all children of the desired parent. If large nests were implemented, then querying of the database would also need to include more complex filtering to find the desired data.

The data structure used had minimal nesting, and was kept to a standardized format, and the format was also continued to be used in the uplink and backend. It was created two main keys (called discrete node) reflecting the simplified node test setup detailed in 2.3 Node and the Gateway, giving one nesting level. The NodeID was an identifier of each discrete node (main keys) in the data structure. It was used to identify and separate the various nodes in the system and was stored of the database root. While the children consist of all the data from a given sensor system at a given time, where each value is represented by a name (the key) and a corresponding continuous sensor value. The two data structures used to store the data by

the sensor system, was a simplified version and an extended version, which is visualized below:

The simplified version:

```
{
        "nodeID": {
                "batteryPower": VALUE,
                "doorState": VALUE,
                "temp": VALUE
         }
}
```

Extended version:

```
{
        "nodeID": {
                "batteryPower": VALUE,
                 "doorState": VALUE,
                 "light": VALUE
                 "noise": VALUE
                 "rssi": VALUE
         "temp": VALUE
         }
}
```

The simplified version was used during the test phases of development (where only dev cards were accessible), while the extended version was created for the fully implemented end system. The result with data is reflected in Database

2.5.3 Design

Firebase was set up as detailed in 7.1.7.1 Firebase Quick Start Guide. The database was tested for open read and write privileges during development for ease of access. All the database rule changes were tested and verified within the integrated Rules Playground. This simulator was used with various operations to ensure the desired functionality and expected behaviour of the database. The simulator was tested with the functionalities: read, set, and update data.

During the first stages of development read and write were always kept open, allowing all prototypes of uplink, backend, and frontend easy access. The security risks were deemed to minimal because the data on the database was mostly dummy data with low personal significance. Furthermore, as the free database plan was used there were no financial risks either, as the free database plan did not include payment of too many requests or downloads. Despite few (if any) security implications of allowing open read/write privileges during development, as per best practice the database security rules were set such that when the group did not develop no one could access the database (meaning the rules were set to false), while during development read and write privileges was activated for all (meaning the rules were set to true). The following rules were thus used during development:

```
{
  "rules": {
    ".read": "true",
    ".write": "true"
  }
}
```

All the functions required for basic database REST communication were manually tested using Postman. A typical usage for any microcontroller attempting communication is the PUT command, which is a basic CRUD operation. The PUT command will create or replace a resource at the targeted resource with the requested payload[33]. Upon a successful PUT firebase would respond with a 200 (OK) HTTP-status, which is an acknowledgement that the data was successfully written. The data was then compared on the client side to ensure if a zero BER (Bit Error Rate) on return was ensured over approximately 50 trials.

This was done both manually and using the pre-defined Postman framework for HTTP CRUD commands. Given a payload of { “nodeID”: “ntnuGateway08”, “doorState”: “1”, “temp”: “27”} in JSON format, and the command CURL. Two flags were raised in the request, “-x” and -d”. The following code was executed:

curl - X put -d ‘{ “nodeID”: “ntnuGateway08”, “doorState”: “1”, “temp”: “27”}’ \ ‘ https://frbasewebapprlcrowdsolutions20.firebaseio.com/ntnuGateway08.json’

And received


```
{
   "batteryPower": 100,
   "doorState": 1,
   "temp": 27
}
```


The data was thus confirmed in the acknowledgement and manually compared within the application. It was tested through 50 trials, which gave a 100% success rate and thus a 0% BER rate. This indicated that the uplink should have no issues writing to firebase as REST endpoint.

As an additional proof of concept, a simple way of requesting data through Postman was also tested. By indicating the URL including the root of the project, all or parts of data may be requested by manipulating the URL of the request. Postman had an easy to use UI that simply required the link, and automatically issued the request on behalf of the user. Postman had to be configured with a GET request with an empty body, and JSON format, as shown in the Figure 18: A GET method carried out in Postman

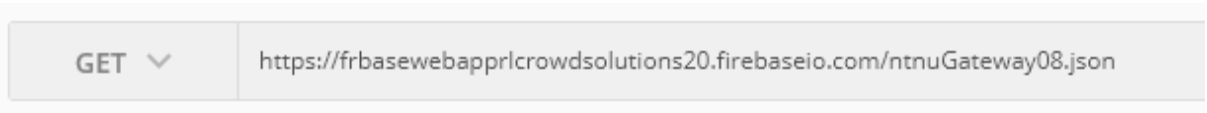


Figure 18: A GET method carried out in Postman

Alternatively, the put command could be executed in Postman, as shown in the figure below:

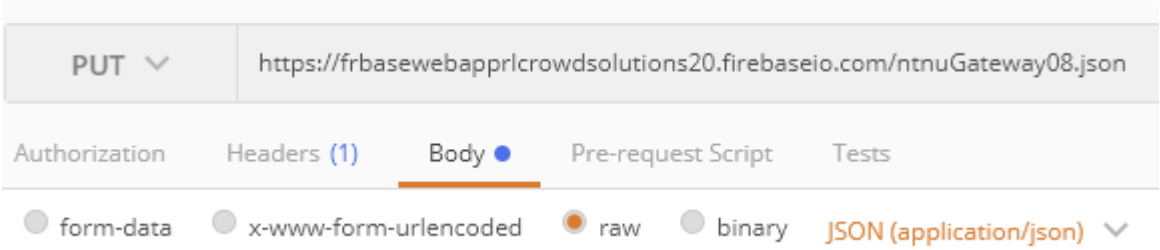
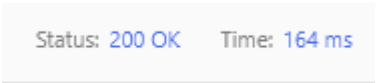


Figure 19: A PUT method carried out in Postman

The body must be configured with the given data formatted in JSON, including the Payload that is desired to be PUT. The user would then be prompted with the response from firebase:


```
{
   "batteryPower": 85,
   "doorState": 1,
   "temp": 17
}
```


The response from the database should in a successful transmission match the payload issued to the database. This would also include a HTTP-status 200 OK and a timestamp (ping) of the roundtrip, as visualized in the figure below. This time (164 ms in the example below) also showed that the communication was at a satisfactory response times for a real time system. Although Postman is not a complete solution for IoT, it is recommended as a testing tool during development. It features a true and tested test environment that is easy to pick up and implement into a testing chain for RESTful API development, as well as



The firebase security rules were improved as real data from the sensor system was uploaded to the database. The real time database incorporates support for the google auth configured for the gateway in 2.4 Uplink. These security rules include the option of authentication for

writing privileges. This implies that the device requesting access must successfully authenticate, with the given database to write to the REST endpoint (such as HTTP(S): PUT or PUSH). However, any device may read the data (such as HTTP(S), GET). These rules were furthermore configured and tested in the Rules Playground until satisfactory performance were achieved. The result, with a total of 62 allows, 0 errors and 0 denies are highlighted in the Database

```
{
  "rules": {
    ".read": "true",
    ".write": "auth != null"
  }
}
```

2.6 Backend and Frontend

2.6.1 Purpose

The backend was designed to serve a real time connection between the database and the frontend. It was designed to be able to serve the frontend with web content (HTML). Where the web content served includes a real time representation, as close as possible, of the sensor data acquired from the database. The backend was designed to be easily accessible by the frontend and introduce as little as possible time latency before posting the updated data from firebase on the webpage. The backend has been designed to forward new data only on update. Meaning the backend was designed with a non-polling (interrupt based) architecture, which reduced unnecessary stress on the database. The backend also processes the raw sensor data, before forwarding the results via a real time socket to the frontend.

The frontend was designed to maintain a real time connection with the backend. The webpage was designed to be user friendly and a simple interface during development and production. The fronted was designed to communicate with the backend through a real time socket, which were used to update the sensor data displayed. It introduced an index page (or home page) featuring the links to all the discrete subpages. Whereas all nodes on the databased were given their own discrete subpage. These subpages displayed the relevant sensor data for the given node. The page was intended as a proof of concept, and it was not focused on designing the nicest web page.

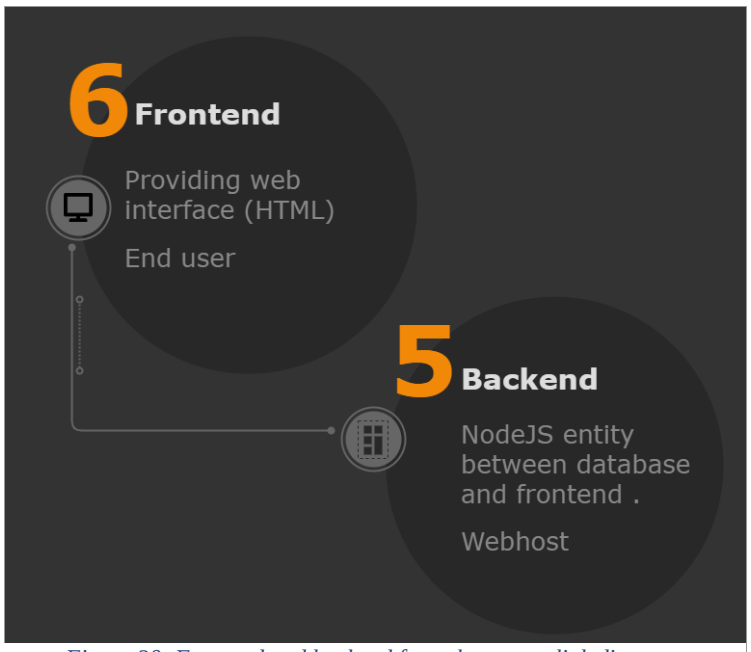


Figure 20: Frontend and backend from the system link diagram.

2.6.2 Architecture

The architecture was designed to consist of three conceptual modules, as shown in Figure 21: Overview of dataflow and modules in the backend and frontend. The backend consisted of *firebase downlink* and *server*. While the frontend consists of a singular conceptual module, namely *front end*.

The *firebase downlink* establishes a real time connection to the database and processes the data. The processed data was published on a channel and was made available to other modules. This gave the other modules access to the processed data. The *server* module listened to this channel and fetched the processed data and forwarded it to the *front end*. The dedicated channel between *firebase downlink* and *server* consists of a publisher and a listener. *Firebase downlink* publishes sensor data when firebase sends a new update of data, while the *server* listens for new updates. The channel is created by a dedicated JavaScript package. All firebase messages are sent on the internal channel, which makes it easy to scale up the number of sensors which are transmitter.

The *server* and the *frontend* interfaced through a socket package. The JavaScript channel has the advantage of being a local channel between JavaScript modules. This channel is thus shielded within the server from any other programs and listeners. The socket on the other hand could be made available to other computers. These communication methods, channels vs. sockets, were thus designed for two different scopes, one being internal and one having the option of being available through the internet.

The web page was only made available within localhost. However, as of design philosophy and future proofing, adding options for handling web pages on the internet was desired. The server could serve custom URL paths with response and requests to web pages on domains. The architecture was furthermore designed to satisfy many of the requirements for a full server rollout. This includes having a dynamic port variable, which listens to the assigned environmental port of a server system. Every NodeID was given a unique web page consisting of a graphical visualisation of the current sensor data.

The sockets on the server interfaced with the frontend through a real time socket. The sockets take the IP of the client and establishes a connection with the server. The IP is defined and

gathered from the clients (HTML files). The client is not required to know the IP of the server to establish the connection. Every web page was given a dedicated socket which will listen on a specific message ID, which corelates with to the NodeID.

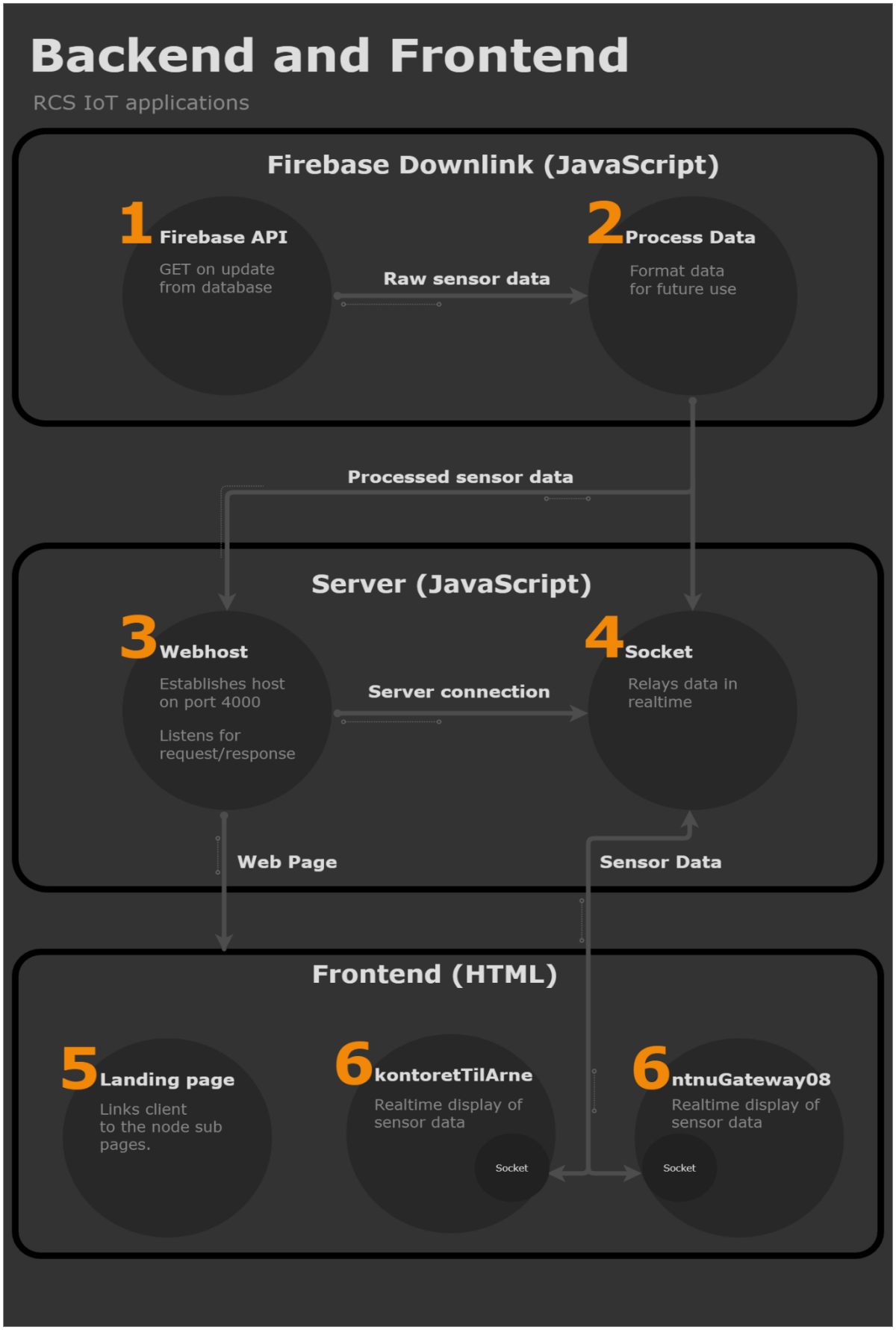


Figure 21: Overview of dataflow and modules in the backend and frontend.

2.6.2.1 Sequences

The system was designed to satisfy two different use cases. The initialization case and the update data case. These cases were further used when designing functions. The sequence diagrams feature conceptual steps which in conjunction makes up the functionality of the code. These steps showcase the events which summarizes the flow of the code.

2.6.2.2 Sequence (init phase)

The initialization case consisted of getting the current value of firebase and posting it on the web page. Figure 22: Overview of sequences in case of initialization for backend and frontend.displays a sequence diagram showcasing the case.

Firstly, the initialization stage needed to be prompted, meaning *connect + auth*. This consisted of establishing a connection between Firebase Realtime Database and the program, done through Firebase downlink. The Firebase downlink forwarded an authentication which the database either rejected or accepted. If the connectivity request meets the demands of the address, the authentication and the project handling, then an acknowledgment was issued. The request would be denied if the server URL were false, unreachable, falsely formatted or wrongly authentication. The system was set to open read privileges which allows any entity to read the database. This made denial of connectivity a non-issue for data fetching, such as HTTP(S) GET. The open read privileges apply only when the connectivity key is accepted, meaning anyone with the key can access the database. More security measures were evaluated, however was deemed outside the scope of this thesis. If connection was not to be established, then the system would halt, and the firebase link would shut down.

The *data GET* would be triggered upon a successful acknowledgement of connectivity in the Firebase downlink. Firebase would once more check the firebase rules to either deny or allow the operation. If the request was allowed, then firebase would respond with the newest sensor data. The data would thereafter be forwarded to a processing step. The processing step reformats the key-value pairs received from Firebase. It is changed to include NodeIDentifier (i.e. which node the data is from) and more humanly readable children names (e.g batteryPower = battery power). The ID was added due to posting all node data from the Firebase Downlink on one internal channel. The ID allowed for the node data to be published on the correct web page, through filtering the messages on the channel by their ID.

The web host would automatically upon execution initialize the hosting. After starting necessary connections to the web host, a socket was created. The initialization of the socket includes GETing the available values from Firebase and establishing a socket which will connect with the web client. The socket would not publish any information on the socket, until a connection with a web client is confirmed. This indicates that if no client is connected, then no data will be published to the front end. The socket is provided initial sensor data, so that the client would not have to wait for an update from firebase. Executing a manual GET from firebase before the socket, allowed data to be faster available for the web client.

A webhost acquired the HTML content and began to serve on the designated port. The server responded to URL prompts in the web browser by sending a response with the related web content. The web page includes the corresponding socket, which when on connection established a link with the server socket. Upon the established connection the server socket emitted the corresponding data, related to the hard-coded ID within the web Page. Where every page was created for one specific NodeID (e.g. ntnuGateWay08).

The web client socket acknowledges the data by responding the identical data back. Upon completion the socket remained active and awaiting new data from the server socket, same as the Firebase downlink. The awaiting of new data would in practice take place asynchronously due to the frontend and backend running on separately from each other. All functionality of hosting the web page and awaiting data on the web host ran asynchronously, meaning the web page could be hosted and at the same time receive new data. This was one of the advantages of using Node.js as further detailed in 2.6.3 Languages and extensions.

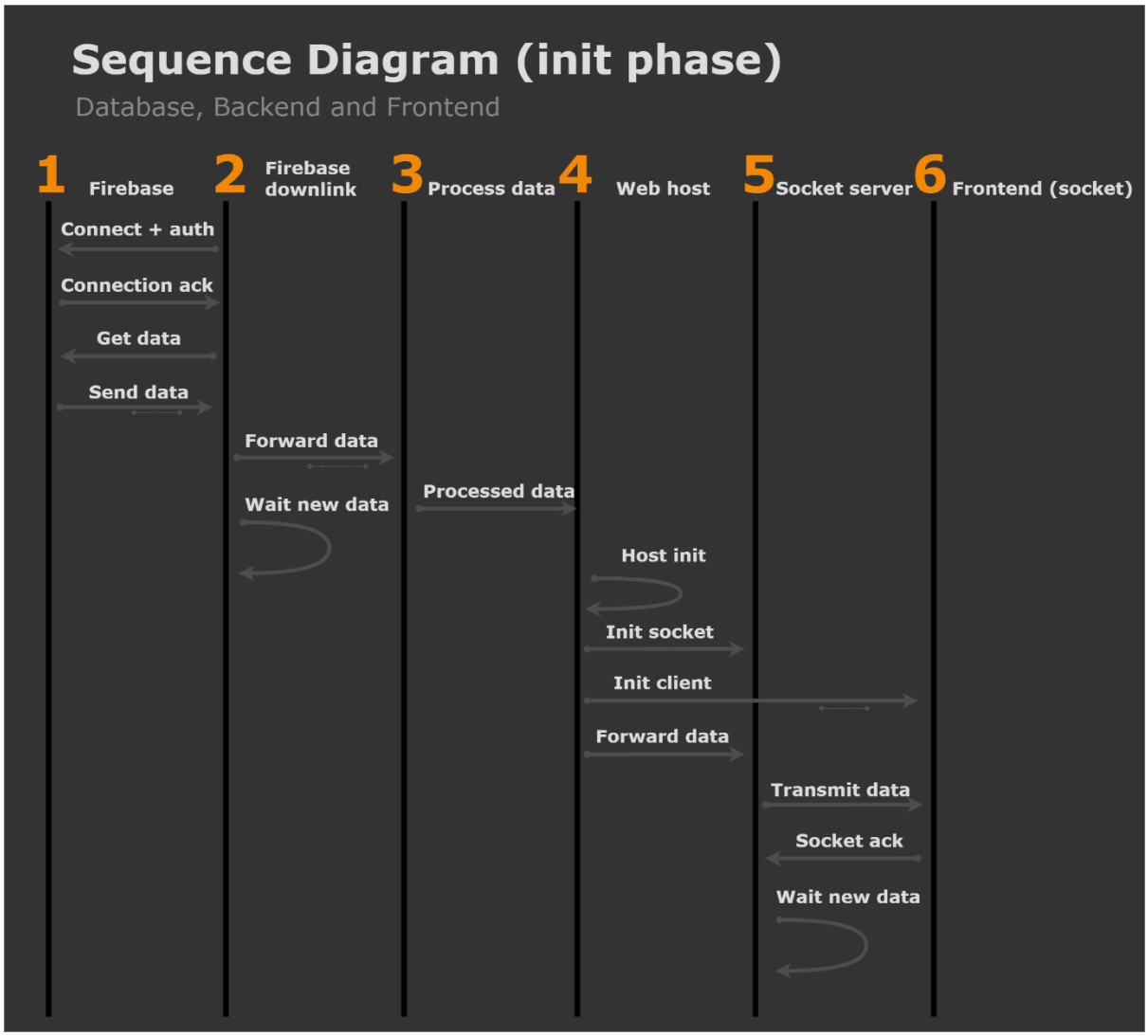


Figure 22: Overview of sequences in case of initialization for backend and frontend.

2.6.2.3 Sequence Diagram - update phase

The update data sequence is the case after the initialization phase, where the system received new data from firebase to be posted on the web page. As initialization has already been performed, the complexity of the case has been reduced. It followed the same flow as the initialization case, however only on data update. The sequence diagram is shown in Figure 23.

Firebase API would notify the Firebase downlink of updated data on firebase. This would first start the process of processing the updated data before it would be published on the channel. The web host would listen to the channel for data and emit the message to the client socket. The client socket would receive the data and update the tables visualizing the sensor data.

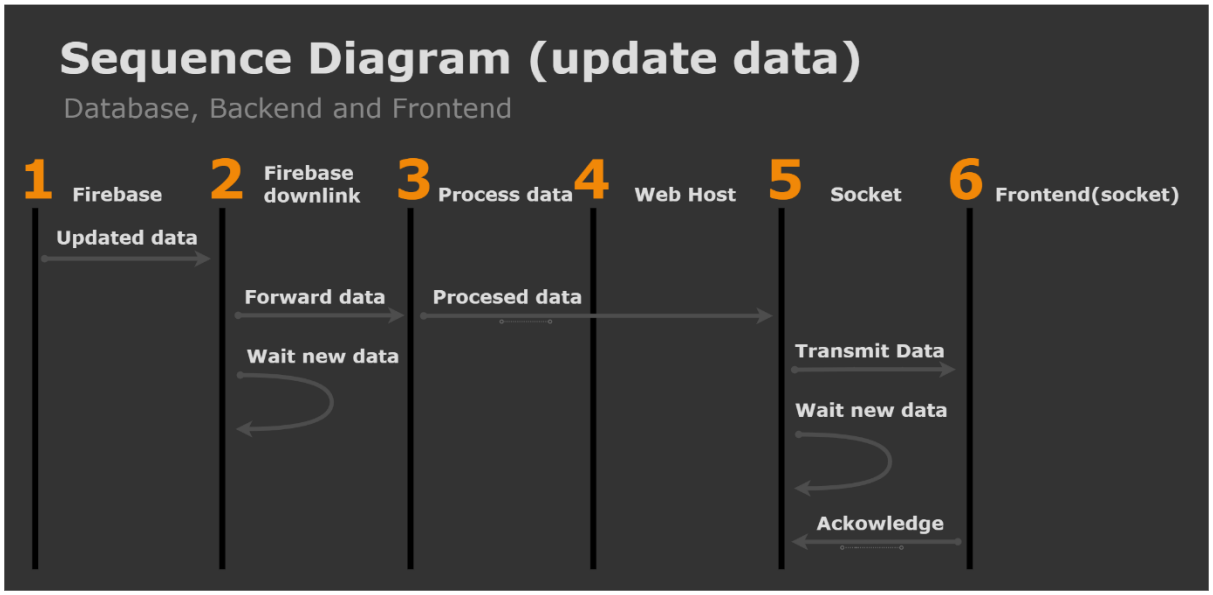


Figure 23: Overview of sequences in case of update data for backend and frontend.

2.6.2.4 Class Diagram

The sequence diagrams Figure 22, Figure 23 and data flow chart Figure 21 was used to design the functions and variables needed in the various modules. The class diagram in Figure 24 presents the modules used in the program and the main functions and variables.

The *firebaseLink.js* module was part of the backend and consisted of 2) and 3) in the sequence diagram Figure 23. It consisted of objects used for:

- Connection to the firebase, named *var firebase*.
- Authentication and for security measures, named var *admin account* and *serviceAccount*.

The process of connecting to Firebase was carried through by the *firebase.initializeApp().*The function *fetchData()* was used to listen for updates from the Firebase API. Where the *processData(data)* was called on by *fetchData()* and the processed data was published to the channel by the function *process.send(),* which is further explained in the class diagram.

The *app.js* module was part of the backend and consisted of 4) and 5) in the sequence diagram Figure 23. It consisted of several variables/objects, which were used to:

- Listen on the channel: *const cp* and *const forked*.
- Host the web page: *const express, const app, const path, const server* and *const port*.
- Setup socket: *const io, const server* and *const port*.

The socket was set up using *initClientData(),* which first listens on the channel for new data using forked.on() and transmitting it to client through the socket using io.on (forked.on() and io.on() which is futher illustrated in the source code). The web host was set up using setUpHost() and the port was set up by setUpPort(). To emit new Firebase data the update client was used with the same logic on updateClientData().

The *web client* consists of three web pages, *index.html*, *ntnuGateway08.html* and *kontoretTilArne.html*. The HTML-file *index.html* only consisted of links to the other web pages and was the homepage. The HTML-file *ntnuGateWay08.html* and *kontoretTilArne.html* offers similar functionality and only differs by the NodeID they identify as. They use socket.on() to receive data from the web host and acknowledges by socket.emit() which will be received by app.js through socket.on() (forked.on(), socket.emit() and io.on() is futher explained in 2.6.3 Languages and extensions) .

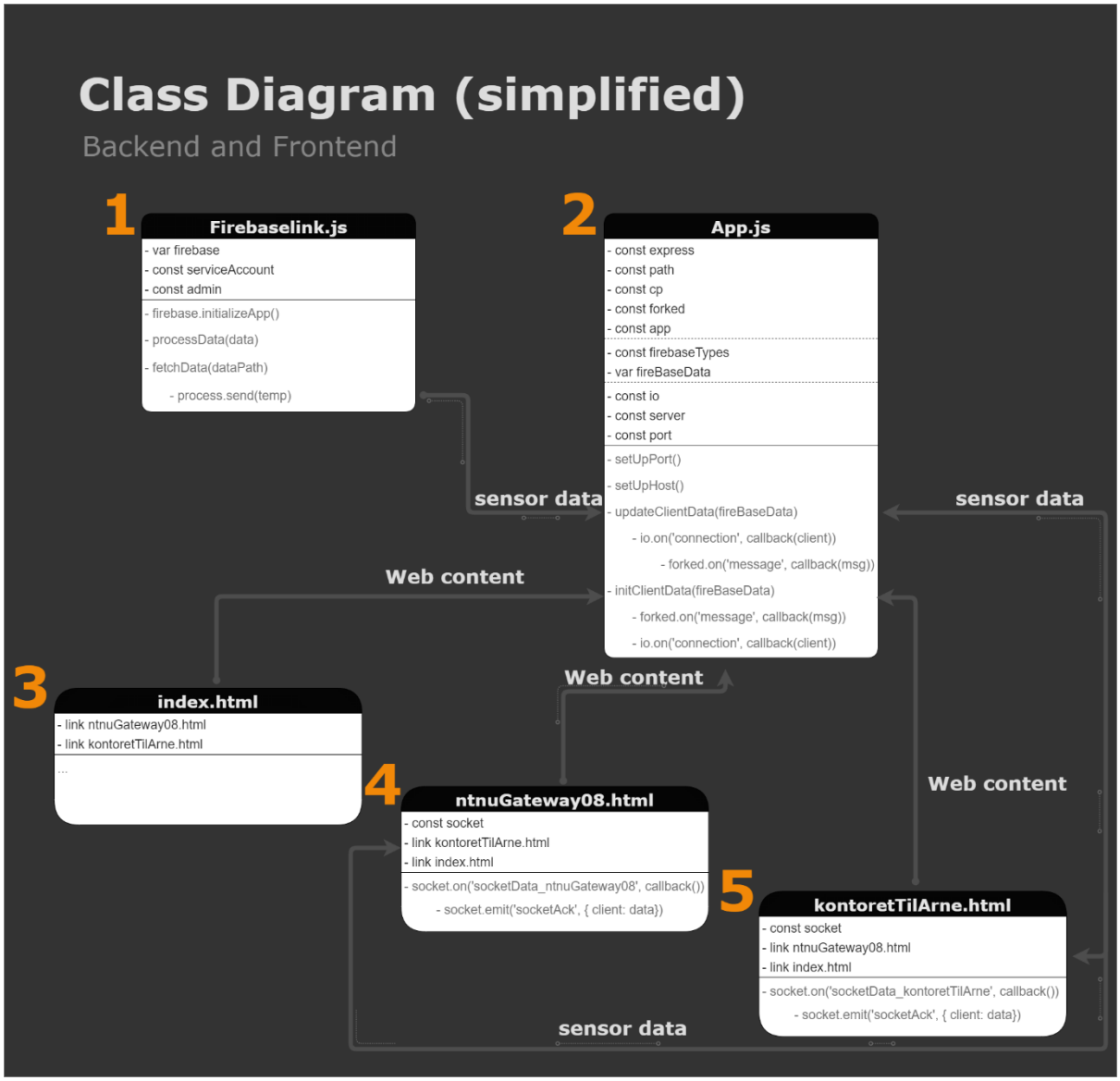


Figure 24: Overview of objects and functions in the backend and frontend.

2.6.3 Languages and extensions

The backend and frontend were made up of several modules, extensions, and programming languages. The modules described in Figure 24 also constitutes the modules used in the final application. Several languages and extensions were used during the development. The languages that made up the final application are JavaScript (Node.js) and HTML. JavaScript was the language used in the backend while HTML was used to mark-up the Web Page. HTML is the standard mark-up language in frontend design and was thus a natural choice. JavaScript was used in conjunction with HTML to make dynamic web pages, as HTML alone remained static. Which is why the modules 4) and 5) are connected to module 2) through sockets in JavaScript, while module 1) and 2) are connected through a JavaScript channel.

2.4.3.1 Node.js

The JavaScript was executed within Node.js, which is built on the Chrome V8 JavaScript engine. One of the reasons for selecting Node.js was the included support for 3rd party modules, which is imported using the npm (node package manager). This included the official Firebase module, namely firebase and firebase-admin, which was used to interface with the Firebase API. Official Google support was important for the choice of Node.js and Firebase as frameworks. The node firebase module is widely used and receives numerous updates on a frequent basis, it is also officially recommended to be used in conjunction with Node.js and wise versa.

Node.js runs asynchronously by event driven logic, and is designed to build scalable network solutions, whereas the asynchronous nature allows for a callback structure. The asynchronous structure allows for multiple functions to run simultaneously. This in return allows the sockets and channels to run in parallel without interrupting blocking operations. The callback structure allowed efficient non-polling coding style. Google took advantage of the asynchronous callback structure when implementing the Firebase API, allowing for efficient Node.js pairing. In total this made for a good combination for IoT smart sensor low power system.

The internal channel was created using child_process. The stock module allowed for publishers and listeners to establish intermodular connections within Node.js itself. The Child Process allows Node.js to spawn children related to parents. These children and parents make

up pipes (channels) at which they enable unidirectional downstream communication. Whereas the socket utilized the module socket.io. This module enables real-time bidirectional event based cross platform communication through a socket. It was used in the place of a WebSocket due to its wide support for devises, at the cost of a slightly higher data usage expense, as discussed in 4.2.3 Socket.io vs WebSocket4.2.3 Socket.io vs WebSocket.

The server framework Express.js, was used to establish the server and web host functionalities. The module is the 7th most popular Node.js module (25th May with 6.4M weekly downloads[34]) and is very well documented and tested, which made it a popular server framework.

2.4.3.2 HTML

The frontend was designed with simplicity and functionality in mind (Scandinavian design). HTML was used to mark-up the webpage, and included running of JavaScript, which was used to provide the real time sensor data at low latencies. This allowed the user client to display the sensor data at real time and navigate between the web pages. The pages were made based on three main types of elements:

- A title, which were used for identifying the specific webpage
- A table of contents, giving links to the other webpages
- A table, showing the sensor system data.

Every web page had one dedicated HTML file which marks-up that specific web page.

2.7 Hardware

The decision to make a custom PCB was made after checking for possible breakout boards and their prices and features. No matches were found for a reasonable price or the required features. The choice was then made to make a custom PCB with the required sensors and matching the required specifications set by the employer. Dev cards that were considered were as follows: Thingy:52 dev card[35] with a price of $40 (~400 NOK), RN2483 LoRa® Mote[36] with a price of $70 (~700 NOK). Both dev cards have the required sensors and wireless capabilities; however, their price cannot compete with a custom solution. The node the team made has a unit price of about $20 (~200 NOK) in low quantities, which makes it a clear choice.

2.7.1 Specifications

The solution that was opted for has the following qualities and functionalities; It can monitor up to 6 switches, which can be used for either doors or windows. It can monitor the temperature in the room as well as the current illumination level with high precision. It can measure the environmental noise in the room it is situated. It should also be cheap to manufacture and produce.

It is powered by a CR2032 coin battery cell and should be able to last for several years on a single charge, the reason for this choice is because it is easy to implement since it has a relatively low footprint, it's the most common coin cell battery and it is inexpensive. The PCB also has a built-in antenna, which is constructed in accordance with the DN038 antenna[37], the MCU is a CC1312R[3]. It has an 8 position DIP switch module and four potentiometers for calibration of the different sensors onboard. It has one 48MHz crystal oscillator and one 32,768 kHz crystal, it includes an JTAG interface and four mounting holes.

2.7.2 Design

The first prototype was created in Eagle from Autodesk®, it has excellent functionality and support. It is also the program that this group's lead PCB designer is most familiar with. The main project was however designed in Altium®, which is an industry standard. The program is very good, but it has some difficulty curve.

The project was split into several documents to make it more legible and easier to understand. The node layout was then based upon the prototype, but with added components and smaller footprints.

2.7.3 Node circuit

The brightness sensor consists of three wires and utilizes the I^2C standard which only uses two wires, the third is an interrupt, this allows the MCU to rest between samples and the sensor can wake the MCU up when it's done with the measurement. The circuit consists of three pulldown resistors and the sensor combined with a decoupling capacitor.

The indication LEDs consist of three LEDs each in series with a resistor, they are not connected to ground, but to a header which needs to be shorted to ground. This makes it possible to disable it to save power without the need of any switches, this saves valuable IO ports and makes it easy to debug without needing to change the program.

The battery is a common CR2032; in our circuit which is low power, the usual capacity of the battery is 220mAh, to avoid any voltage spikes the positive pole has a ferrite coil with about 1,5kΩ resistance to HF (100MHz). Since our application is subGHz this works well to stabilize the voltage from the battery, this is important since the battery has a big metallic surface area and is very susceptible to noise generated by the HF circuit. This would cause quite the problem when sending data, if the voltage starts spiking, then the MCU might start malfunctioning or even worse get fried and this in the middle of a transmission. See Figure 47.

The JTAG connector is wired identical to the CC1312R LAUNCHPAD XL, this will make it very easy to program, since you can just connect the JTAG directly to the programmer port of the Launchpad. The JTAG utilizes the 1,27mm spacing, this makes it more compact and you will not need to manually connect each wire, but just connect a cable. See Figure 54.

Three of the potentiometers are connected as a voltage divider with a decoupling capacitor between ground and the measurement pin, the other potentiometer is used as an adjustable gain resistor for the decibel sensor. See Figure 58, Figure 59 and Figure 60.

The decibel sensor is essentially a microphone with a few adjustments; firstly, it can be enabled by a transistor and has two filters, one DC filter and one lowpass filter. The DC filter will block the DC while the lowpass filter is designed to let frequencies below 2 kHz through and dampen everything above this threshold with -20dB/decade. The signal is then amplified by an operation amplifier whose gain can be adjusted by a potentiometer. The resulting signal can selectively be inverted or kept as-is by soldering the bypass resistor or the inverter circuit. See Figure 64, Figure 65 and Figure 66.

The synchronization circuit was made with the intention of making it user proof. Even if you connect it with the wrong orientation, it should not break either of the devices however, it should detect the wrong polarity and notify the user through the indication LEDs. This circuit was completed by using a 5 pin design and implementing an NPN transistor to ensure no overvoltage damage from incoming pulses from the Master node, this can happen because of the decreasing voltage from the battery in the node, the MCU cannot receive any voltage above 0,3V higher than V_{cc}.

The fix to this problem is to place the ground in the middle pin three and V_{cc} on the fifth pin, data pins on two and four and an additional ground on the first pin, the Master node will know that the circuit is correct if the sense pin is low, and incorrect if it's high. The NPN transistor makes the signal coming from the Master node adapt to the battery voltage provided from the node if it is reversed it will simply not do anything which is our goal. In the final product physical barriers will make it impossible to orient it the wrong way.

The reference design provided from the manufacturer makes the central components of the node easy to implement, since values are already calculated, but anyway some values must be calculated to ensure correct operation.

Built into the MCU is a DC/DC converter, which makes the voltage used by the RF circuit, this circuit was left untouched since the design is sound and requires no altering to suit the needs. The crystal oscillators were changed for convenience of distributor and price. This required the recalculation of the load capacitors, which were calculated to be 20pF. The RF circuit were left as is, except for the impedance match; which will be tested and matched after an analysis with a network analyser.

2.7.4 Node PCB design

Early prototyping was conducted with a shaven down reference design from the manufacturer, this was paired with a thoroughly tested PCB antenna known as DN038.[37] This antenna has a good enough range and footprint for our usage.

The design was made to accommodate bigger components for ease of hand soldering, components with the imperial size of 0805 were chosen, and this resulted in the reference design having to be altered slightly in some places, more specifically near the crystal clock and the RF DC/DC converter. A simple two-layer PCB was selected with a thickness of 0,6 mm, this is a specification of the selected antenna.

Our second iteration fixed some of the main design issues that arose from the first prototype, most notably the polarity of the crystal oscillator, smaller components 0503 (imperial) and JTAG wiring. A rectangular case was selected for simple case construction.

In addition to the above changes more sensors were also included in this design, main considerations here is: low power, avoid noise propagation between sensors and antenna, accessible adjustment components, ease of soldering and short trace lengths. Because of the previous implementation of the 8bit DIP switch, the traces had to be altered, since the previous design utilized some of the ADC enabled IOs, in this new design there were required four of them. The PCB were kept at a two layer; since the circuit is not power hungry and therefore does not require a power layer, it is neither very noisy; therefore, a big ground plane is not needed.

The decibel sensor was placed in the bottom right corner, since the microphone is sensitive to vibrations that can occur in the board itself, which is why it is placed next to a fastener. The brightness sensor was placed with good clearance to the other components, such that it can be fitted with light isolating material to shield from LEDs on the PCB. The 8bit DIP switch was placed closer to the MCU such that some wiring from the decibel sensor is uninterrupted.

The adjustable potentiometers were placed right below the RF circuit, this may be the cause of some concern, however the RF circuit will never run while a read is in progress, despite this, some via stitching in the area and a decoupling capacitor were added just in case.

The battery was placed in the top left corner where it is accessible without making the PCB unnecessary big. A smaller IO port were added on the left side, the intention here is to make another smaller PCB to interface with this port and will include the screw terminal blocks, this will make it easy to reconnect the node if the connection is somehow severed, since you will not need to physically disconnect all the wires.

The space next to the connector is reserved for the indication LEDs, these will tell if the node synchronization works and if it is connected in the right orientation. The LEDs are disabled or enabled by a terminal header which must be shorted for the indicator LEDs to work. The synchronization header is located on the bottom left corner with enough clearance to be easily slotted onto the Master node for synchronization

JTAG header was placed right next to the synchronization header because the placement was optimal for accessibility, close to this header is the reset button, there is also a via placed on the PCB which is only connected to the remaining pin on the MCU, this was done deliberately since it can be used for debugging.

Trace routing considerations were as follows, power lines are to be routed on the bottom side and the ground plane is to be as uninterrupted as possible, this will stabilize the voltage spikes that may occur. Data lines are for that reason kept as close as possible

2.7.5 Components

Components that were used in the node PCB were mainly selected based on price, but to ensure consistent consistency the same manufacturer for most of the capacitors and resistors were chosen. Some exceptions were made, such as the coil and capacitors used for impedance matching and filters, these components therefore have stricter tolerances; this will increase the effective range as much as possible.

These components were selected based on performance, function and operation, rather than price:

- One ferrite bead
- Three LED’s
- One JTAG header
- One 8bit DIP switch module

- One brightness sensor
- Four potentiometers

The ferrite bead is referenced in the manufacturers recommended layout and is used to filter noise out from the battery, since the battery has a large surface it will gather a lot of noise when the antenna is active and thus needs to be filtered.

The LED's must be able to operate at the MCU's minimum voltage; which is 1,8V they also must be SMD. No parts were found under the target specifications, but a close match was chosen with a typical forward voltage of 2V, which for our intents and purposes is good enough. The JTAG holder needed a shroud, such that the programming interface cannot be inserted in the wrong orientation, an additional requirement was to accommodate a 1.27mm pitch variant for a smaller overall footprint.

The 8bit DIP switch module was selected for compactness; 1,27mm pitch as well as having protruding switches. This modules duty is to provide easy parameter adjustments, the first six bits will enable or disable the door switch reader, while the remaining two switches will control the brightness sensor's sensitivity level and the idle sampling rate.

The microphone on the other hand had some stricter requirements, it had to be able to resist hot air from the soldering process without getting degraded or destroyed by the flux as well as to have a solder friendly layout. And the frequency response must be within our requirements, such that the frequency of typical speech can be detected. The brightness sensor was selected based on the power requirement from the design itself, it must be very power efficient, TI's OTP series lumen sensor was opted for, there were also several other candidates.

Inolux's IN-S series were considered; however, the power draw was too high, up to 1mA with high illuminance, which is simply too much and with the price for a control circuit it would be cheaper going for an IC with everything already integrated.
Lite-On's LTR-3 series were also considered, however their power draw were too high and the operational voltage requirement doesn't match the MCU's operational range which is 1,8 – 3,3V, it's range is 2,2 - 3,6, this isn't sufficient and will either need to compromise on

battery life or disable light readings earlier. Neither of those are anything that can be compromised upon.

ON semiconductor's LA0161CS were also considered, this one had a very good power draw of just 210µA at max, however it suffers of the same supply voltage limits of 2,2V – 3,6V it also does not include an interface to easily read the values and requires an ADC port to read the values, which requires more power.

TI's OTP3006 was excellent for our use, it has a voltage range of 1,6 – 3,6V which is better than our MCU, it's power draw is also excellent with a max draw of 2,5µA. It also sports a hardware interrupt pin, which enables the MCU to rest while the sensor measures the light intensity.

The potentiometers were selected based on space, user-friendliness and price, the spot for the potentiometers were already decided before the parts were decided upon, therefore the footprint had to be accounted for, it has to be easily adjusted and also since the node required two different resistance potentiometers the series that should be selected must exist with different resistances; one with 10k and one with 100k. The one that was opted for were one of the cheapest that had the screw on the top while still being properly insulated, since the space between the potentiometers is very limited.

2.8 Node case

2.8.1 Case 3D design

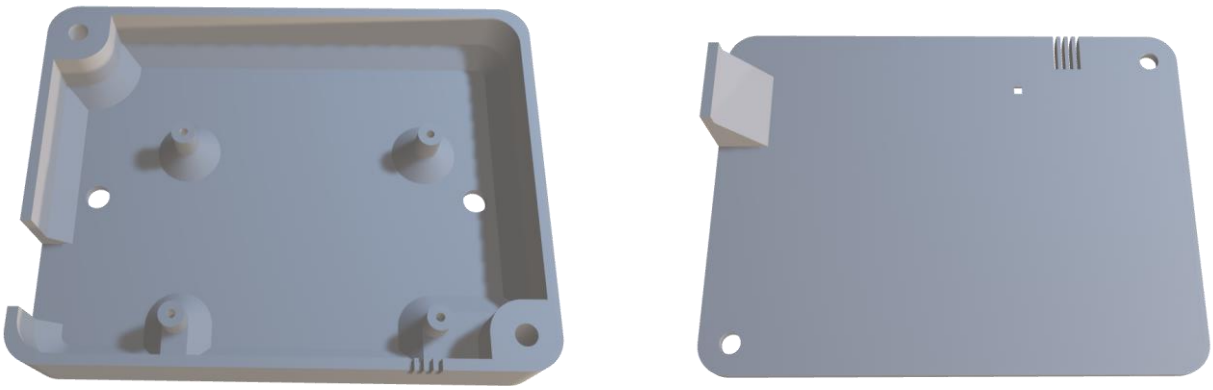
The first draft was sketched in fusion 360, since that is the program the designer was most familiar with. To make the design as accurate as possible, the PCB was imported from Altium through conversion to .dxf format, which is a format that includes multiple 3d models and their position relative to one another.

The case was designed with several key elements; There must be a hole for the brightness sensor, which must be located directly above. There must also be a hole for the microphone; therefore, a square hole with studs was put in place. The case was separated into two parts for convenience, the bottom one will hold the circuit board, while the top part will seal the case.

2.8.2 3D-Print setup

The team modelled and printed a prototype to explore the opportunity of a casing for the sensor node. CURA was chosen as the gcode-converter for our model. The gcode was converted using “draft” setting. This is a quick and easy fix for a prototype model, as a working model could be quickly produced to both improve on and see how it would compare to the envisioned shell of the CC1312-Node.

The “draft” print takes about 3 ½ hour to produce using standard PLA and a standard Anycubic I3 Mega printer, although a better printer could produce it even faster. Increasing the quality does not seem to be worth it at this stage, as it takes considerably more time to produce, and given the opportunity to realize our system as a commercial product, the team would rather use higher quality prints, for example through firms specialized in such production. The quality came out very acceptable considering our requirements for a simple demonstration of our idea, and the team have no need to further the product right now; however, it is likely that it could be improved it in certain aspects given future development of our project.



3D print draft

Figure 25: Bottom

Figure 26: Top

Figure 27: Finished 3d Print, assembled

2.9 Testing the system

Testing the communication link can be seen step by step in 7.3 Testing the system. Here the node data gets transported from the node to the web browser where the data can be observed.

2.10 Testing node life using Energytrace

Energytrace is a tool within the CCS client, for recording energy profiles of different Texas Instrument MCU's. Energytrace is not a measurement tool that can be compared to a lab grade dc power analyser but will give an approximated current and power. Energytrace itself uses energy and therefore the current and power used will be displayed as higher than the real current and power in the board being measured.

Calculating the lifetime of the sleepy nodes in the IoT system is crucial as the lifetime influences the security and reliability of the system. A battery life of about 10 years seems like a reasonable goal; considering similar IoT devices, like wireless door and window sensors, have about a 10-year lifespan. The team considered the mechanical sensor, the reed switch, which has an expected lifetime in terms of operations. This means that the reed switch used in the custom board must have a good life expectancy. As an example, the reed switch used in the TI ULP-Sensor Boosterpack hardware, which is used when developing programs to run on our custom board. This reed switch has a minimum life expectancy of 50 million operations. As an example, 100 operations per day means a life expectancy of 1369 years or 500000 days.

The number of events the node experiences per hour can vary a lot. Regarding the purpose of the design of the Reed switch sensor only node, the team estimated the number of doors opening and closing to be 64 a day, or 64 packets sent per day. As the testing room would be a small meeting room for 4 people at NTNU Gløshaugen. one of these rooms can be booked for 2 hours. If four people entered and exited the room once in this 2-hour period, there is 16 switching of the reed switch if they open and close the door for each member that passes through the door. The meeting room can per day be booked for four 2-hour periods or 8 hours in total, which is where the estimate 64 per day comes from.

Having a timer where every 2 minutes the sensor values were sent from the node, or each time the reed switch changed state; if the reed switch changed state the timer would reset, as such one can approximate a general per hour packet sent. This means that if the reed switch

did not change state and a packet was sent the node would send 30 packets per hour. With the reed switch changing state 8 times per hour as above in the reed switch sensor only node the amount would be at an approximated maximum of less than 38 packets per hour.

Testing the power consumption of the node that would be in the system would be done using Energytrace, as there was no access to lab equipment, due to the Covid-19 pandemic. Step by step procedures for testing the LAUNCHXL-CC1312R launchpad as a node running the wakeonreednodefinal[28] can be seen in 7.4 Testing node life.

3 Results

3.1 Data from the system

The test conditions were a home about 85 m^2 footprint, with a radius of 9 meters between the node and the gateway. The absorption material being either 30 cm of timber or 2 cm of wooden panelling and about 28 cm of isolation. The results showed 20 out of 20 packets sent from the node, were received at the gateway. Which then relayed the sensor data to the uplink, that is the ESP32 Wroom devkit-c, transmitting to the personal home router where the data ended up in the firebase web server. The backend of the system did also work as intended, and real sensor data was observed on the localhost server using a web browser. The latency between a Reed switch interrupt and changes on the localhost server was approximated to be half a second.

3.2 Results of Low power Testing using energy trace

Having tested using a LAUNCHXL-CC1312R launchpad with a mounted ULP sensor Boosterpack as a node, along with another launchpad as a gateway. The Energytrace readings on the node can be found in the energytraceresults.pdf[38] file. This pdf contains the values used when estimating the battery lifespan of the node.

3.2.1 Battery life calculations

The lifespan of the node can only be accurately estimated with multiple tests and a long timer, along with proper lab equipment like a dc analyser. The team could only approximate the node's lifespan, as having the timer, in Energytrace, be longer than 10 seconds would make the energy trace readings unstable. gave readings that were sometimes 30 seconds and sometimes only 14 seconds due to error code “Bad energy trace data was detected in trace stream". Either way, theorising can be done as the team knows the lifespan and energy when no wakeup of the CPU happens.
From the pdf covering the Testing data, the battery lifetime was calculated from the 10 second recording of 1 reed switch event and an immediate packet was sent from node to

gateway.

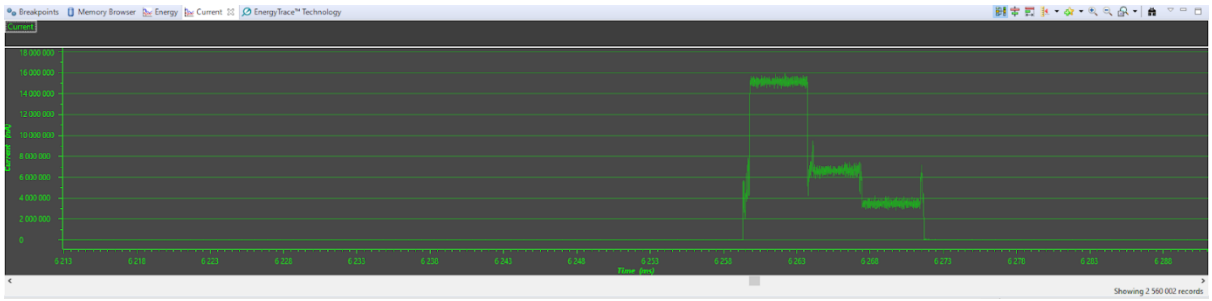


Figure 28: Amperage/time

In this 10 second interval, see Figure 28, 5ms were spent waking up and sending the packet, 5ms waiting for an ack from the gateway and 5ms spent going into low power mode. The rest used in wakeup on sensor-controller interrupt mode where about 1 micro amperage is consumed. Illustrated by the nA/time graph in the pdf document. The current has 3 states where it goes from high to medium to low in the CPU awake state before going back into standby mode. The CR2032 have 235 mAh corresponding to 864 As, which is the coin cell battery to be used in the custom node PCB's. Reading off the different values we get, see Figure 29:

CR2032	864	As
istandby	0,001*E-03	A
tstandby	1350	s
ihigh	0,015	A
thigh	0,005	s
imid	0,007	A
tmid	0,005	s
ilow	0,004	A
tlow	0,005	s

Figure 29: Table of measurements

Using the equation in 1.1.6 battery life calculations, Equation 1. We get an approximated line diagram over events per day to years of life. See Figure 27.

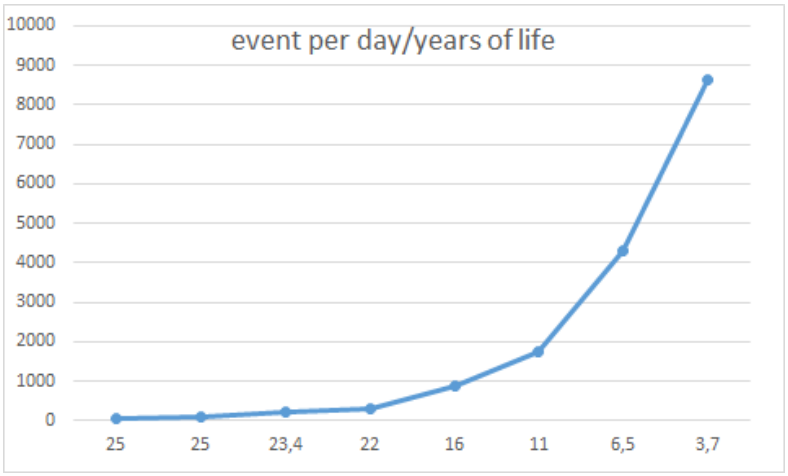


Figure 30: Events per day/life

Figure 30, 100 events per day, 200, 300 and up to every tenth second is shown there. This might not be too far off the real values for battery life as Energy Trace estimates that 1 event every 10 seconds will give the node a lifetime a bit above 2 years. See energyrtaceresults.pdf[38].

It was also calculated in terms of energy, where the group divided a day into 10 second periods. 64 of those periods belong to the reed switch waking up the node CPU and receiving an ack, and the other 8576 periods belong to the standby state, referenced in the figure as toggle and ack and no toggle and no ack respectively. The number of joules in the battery was estimated to be 2350 joules using a CR2032 coin cell battery as reference[39].

			total joule per day
number of 10 seconds per day	joule per 10 sec	Periods	
toggle and ack	0,338*10^-3	64	0,021632
no toggle no ack	0,033*10^-3	8576	0,283008
		total	0,30464
total joule in lifetime of toggle and ack	2350	joule	
amount of days	7714,023109	days	
amount of years	21,13430989	years	

Figure 31: Energy mathematics

In Figure 31, as an approximation, 21 years makes this system truly low power. If the node does not receive an ack from the gateway the lifetime becomes a little different, therefore it was calculated that 50% of the time the gateway does not receive an ack, and it got a lifetime of 9,9 years, only 0,1 years shy of the 10 years which are usual for wireless door and window sensor systems. This is not a surprising result since the node does not get an ack it will wait a certain time and send up to 2 more packets increasing its CPU awake time and therefore its power consumption. The power consumption with a wakeup and no ack from the gateway in ten seconds is 11,137mJ versus the low power 0,338 mJ when an ack is received.

The micreedlightnodefinal project does not work, because of a bug making sensor-data connected to the ADC set to 0 on the gateway. Testing on its energy profile was still done, in the pdf document, as a comparison to the wakeonreednodefinal[28] projects low power consumption. The lifespan was just 17 days for a battery source like the CR2032 battery. Therefore, we took out the 2-minute timer and the power consumption went down dramatically. No wake up and no ack from the gateway, on the wakeonreednodefinal project was 0,033mJ, on the no 2 minute timer micreedlightnodefinal it was 0,848mJ. And with only 64 wakeups of the CPU per day we found out that the lifetime of the node was approximately 321 days. Which is reasonable for a system with a reed switch and 2X12 bit ADC's. We also did as in the wakonreednodefinal project and thought about 50% no ack received and this brought our battery life down to 306 days.

3.4 Database

The results showcase the allowed and denied connections attempted to the database, as well as the sensor data stored in the database. As featured on Figure 32 and Figure 33

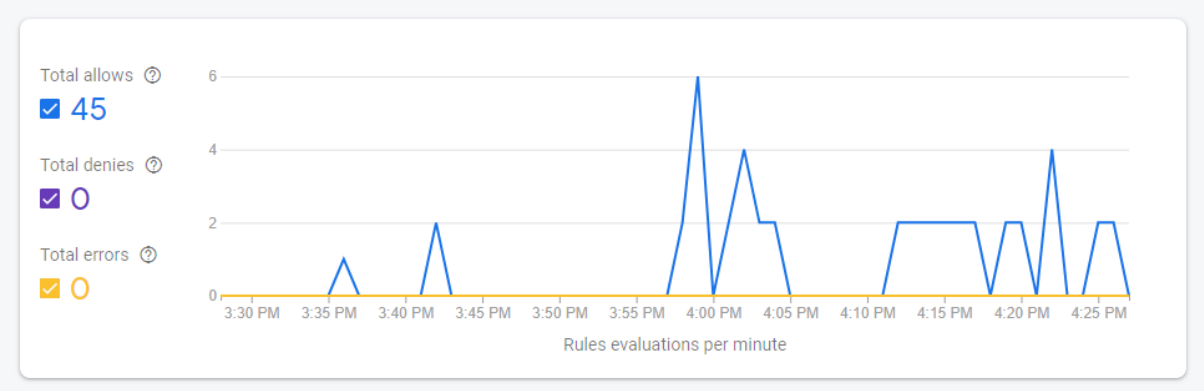


Figure 32: Number of Allowed, denied and total errors during an hour of system stress testing.

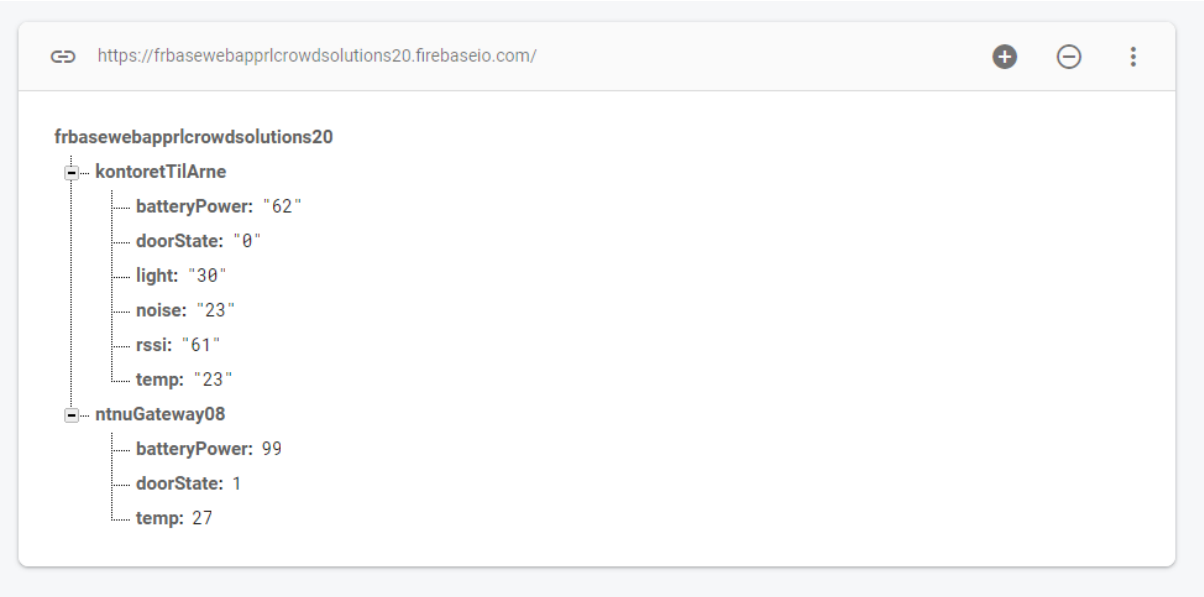


Figure 33: The final data gathered during an hour of stress testing.

3.5 Backend

The initialization and update case of the backend is displayed in Figure 34. The terminal history is a result of executing app.js on Microsoft PowerShell inside VS code. The server was connected on port 4000 and interacts with a client through socket.io, as seen on 3.6 Frontend. Several prints were made inside the code to the terminal to aid in debugging and development. The client confirms the sensor data through an ack, as seen in the terminal. *Initial gathering from database at start* indicates that the child process fireBaseLink.js has been summoned (term used in the documentation). The *undefined data* indicates that a GET from Firebase is still in progress, while *Payload Requested…* indicates that a Firebase GET process has been prompted. While *client:{…. Some Value ….}* displays the acknowledgement from the client.

```
[nodemon] starting `node app.js`
Listening on port 4000...
kontoretTilArne undefined
Initial gathering data from fireBaseLink at start
ntnuGateway08 undefined
Initial gathering data from fireBaseLink at start
{ client: { Loading: '...' } }
Process startup
Payload Requested...
Payload secured and formated!
Gathering data from fireBaseLink when changed
Payload Requested...
Payload secured and formated!
Gathering data from fireBaseLink when changed
{
  client: {
    'battery power ': '62',
    'door state ': '0',
    'light ': '30',
    'noise ': '23',
    'rssi ': '61',
    'temp ': '23'
  }
}
kontoretTilArne undefined
Initial gathering data from fireBaseLink at start
ntnuGateway08 undefined
Initial gathering data from fireBaseLink at start
{ client: { 'battery power ': 99, 'door state ': 1, 'temp ': 27 } }
```

Figure 34: A screenshot of the terminal during normal operation. The terminal displays both the initializing case and the update case. The client returns the data as an acknowledgement, that can be seen on the printout.

3.6 Frontend

The results in frontend displays what the end client will display. The pages are accessed on Google Chrome and may be seen on Figure 35, Figure 36, Figure 37

Figure 35: The index.html accessed through Google Chrome on localhost port 4000.

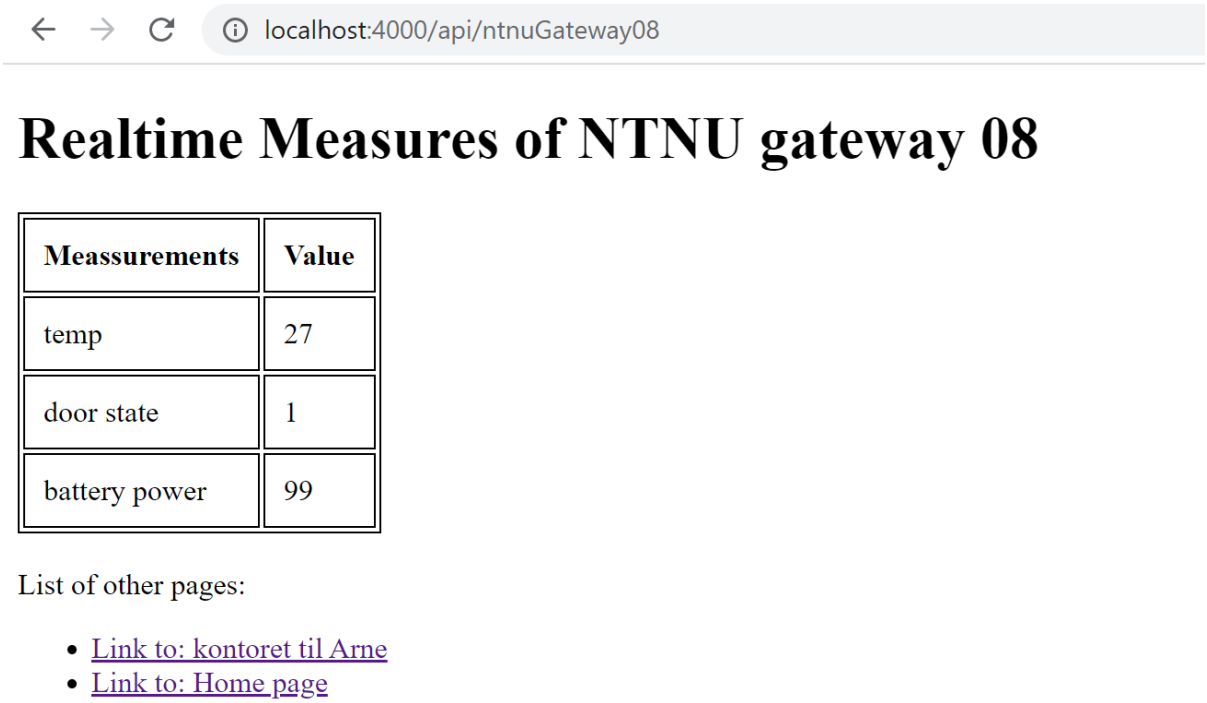
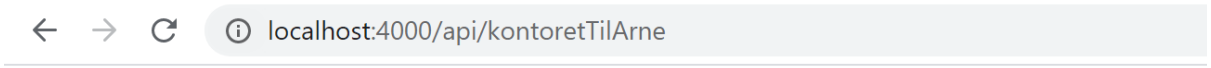


Figure 36: The ntnuGateway08.html accessed through Google Chrome. The page displays real time sensor data from a real-world test of the IoT application.



Realtime Measures of kontoret til Arne

Meassurements	Value
temp	23
rssi	61
noise	23
light	30
door state	0
battery power	62

List of other pages:

- Link to: NTNU gateway
- Link to: Home page

Figure 37: The kontoretTilArne.html accessed through Google Chrome. The page displays real time sensor data from a real world test of the IoT application.

3.7 Hardware

3.7.1 Node architecture

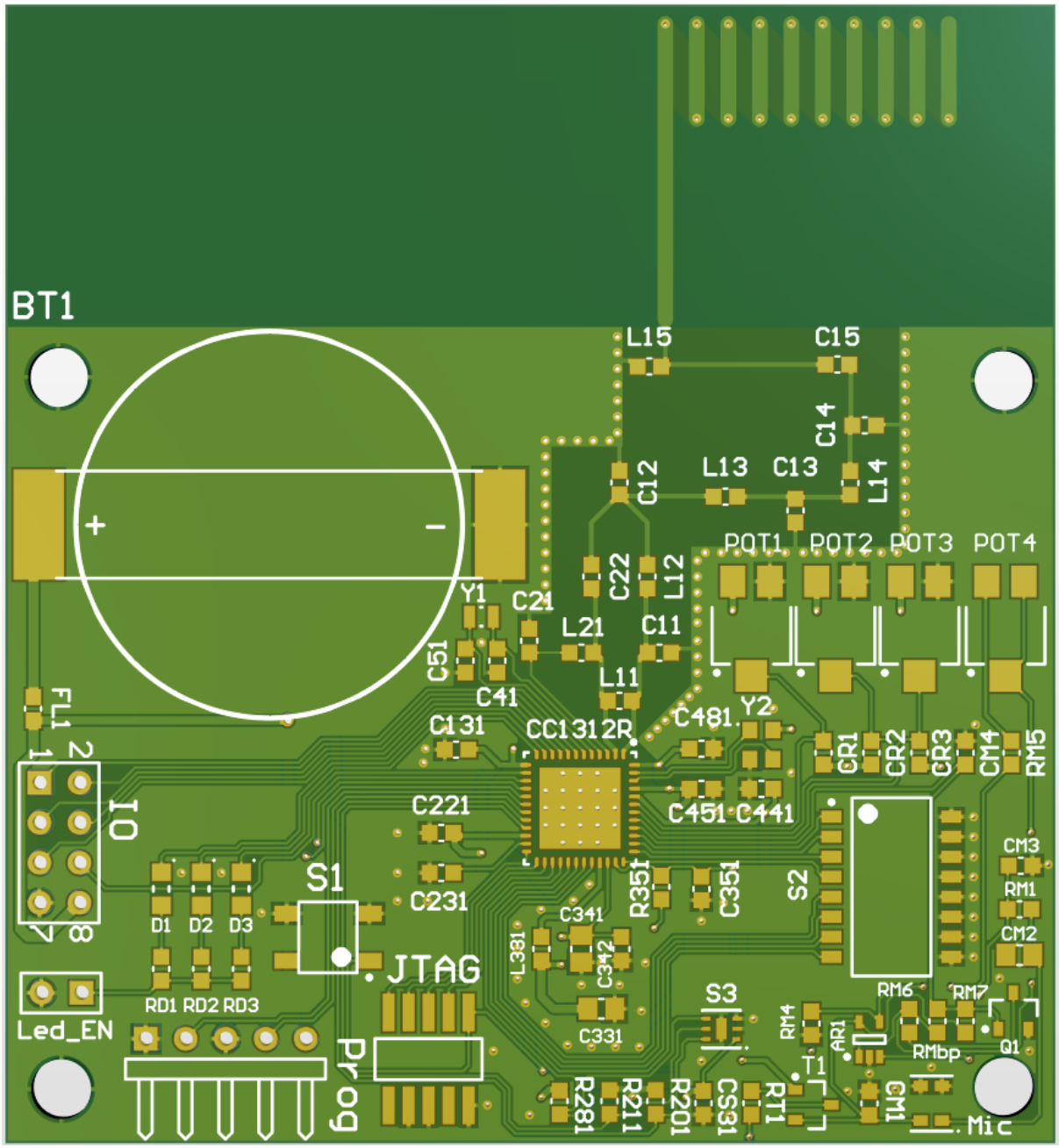


Figure 38: Layout top side

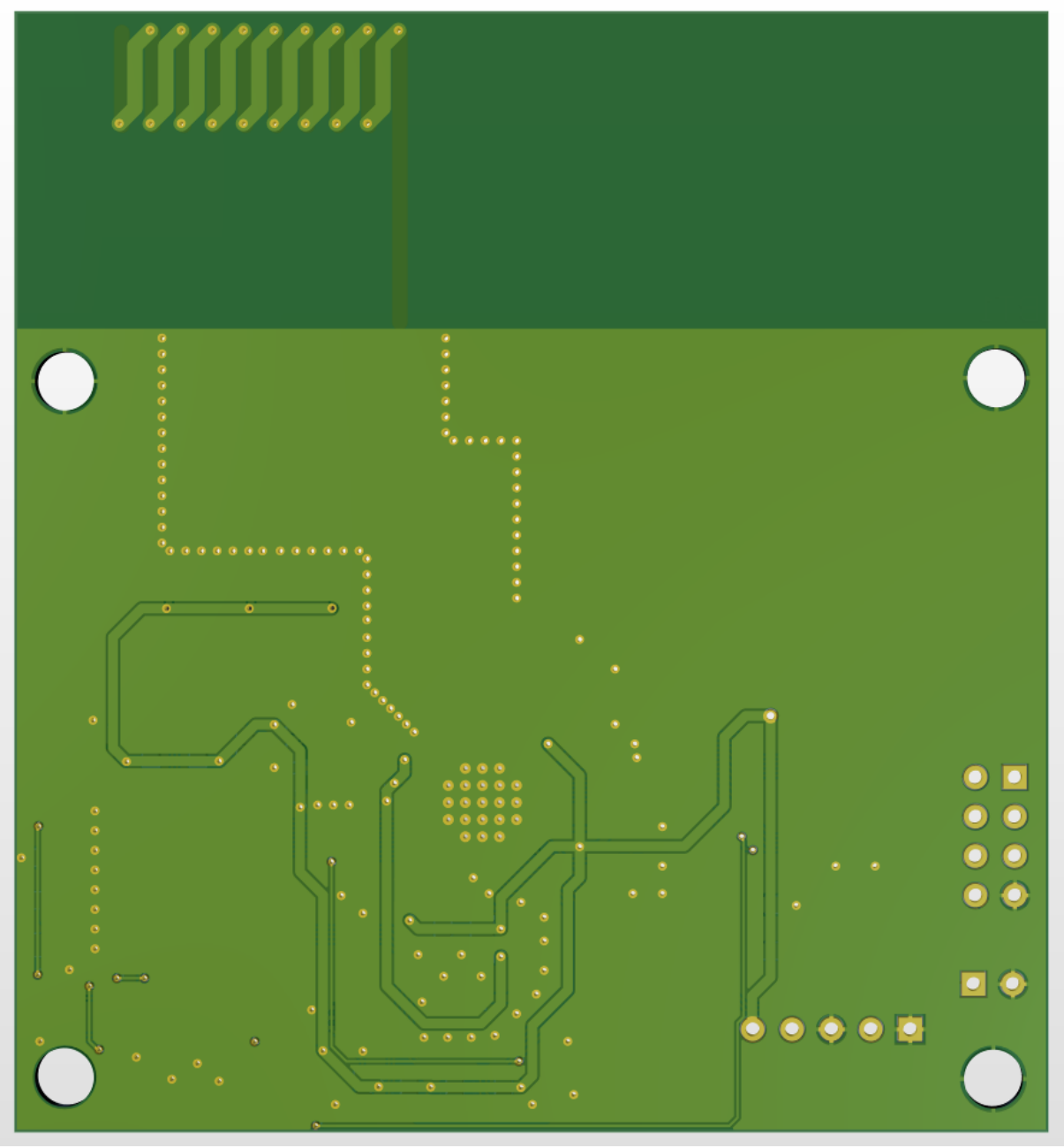


Figure 39: Layout bottom side

3.7.2 Node PCB

This is the final prototype, unfortunately the team lacked the required equipment to solder on the components to form a functioning node

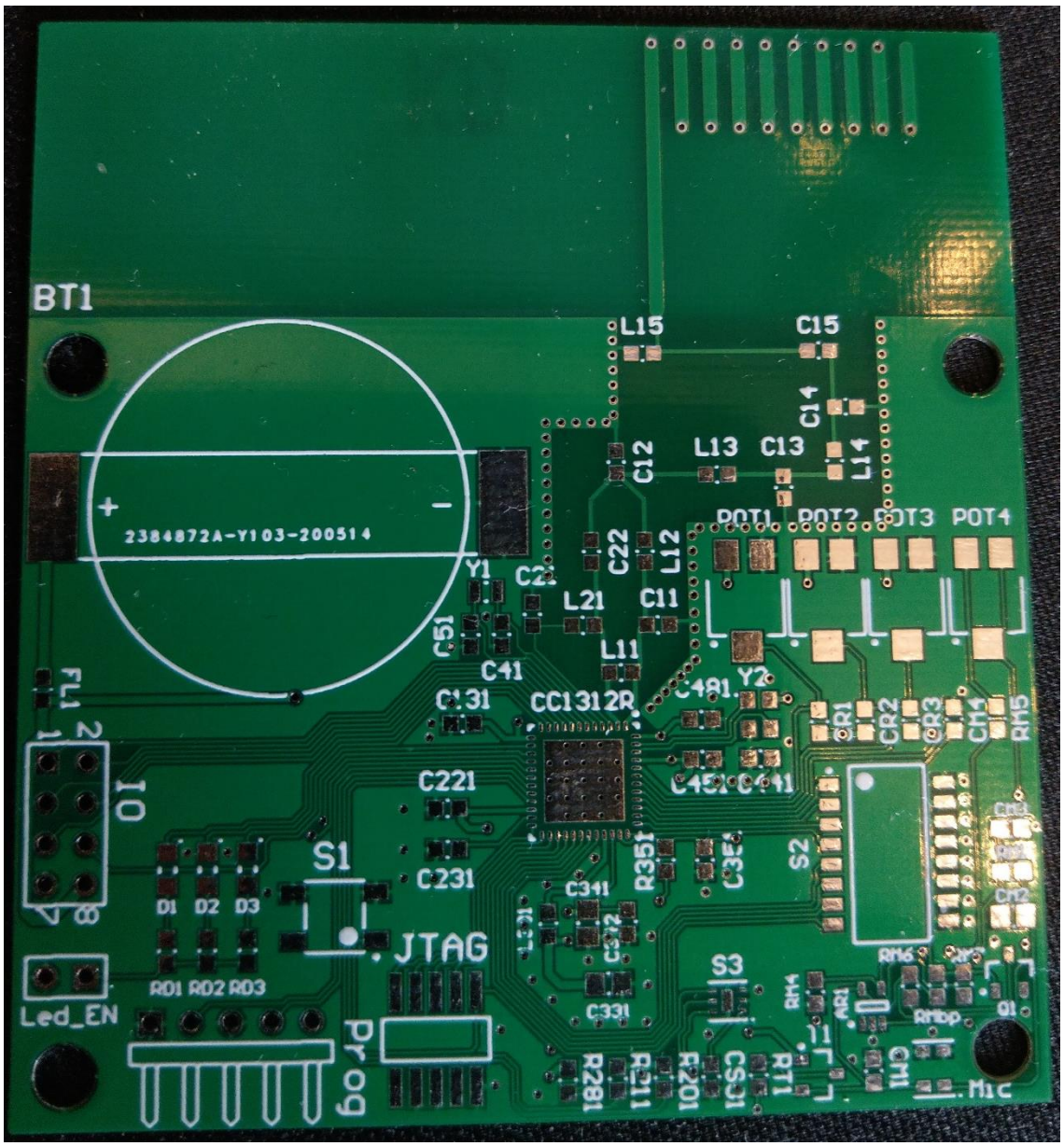


Figure 40: PCB top side

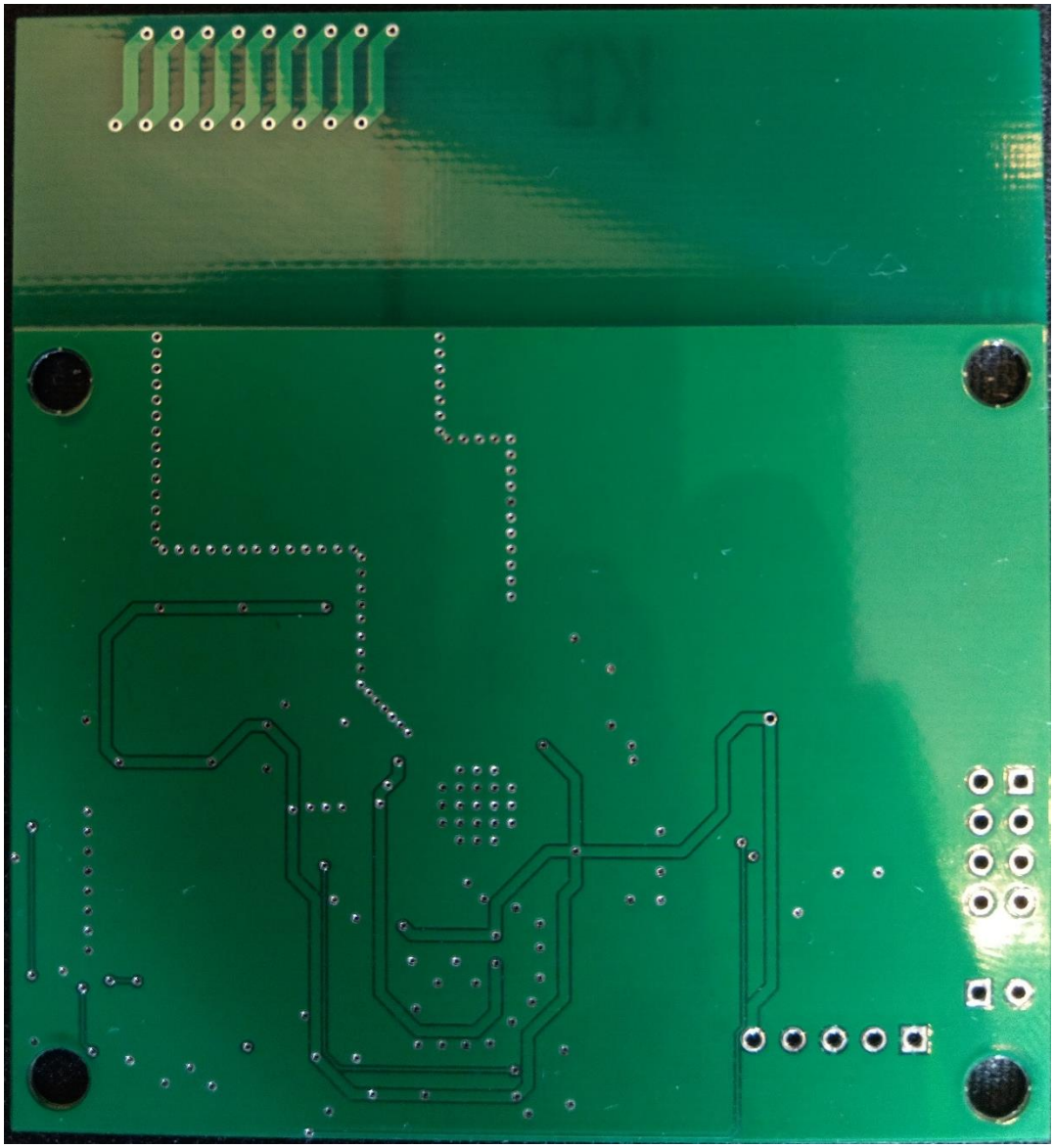


Figure 41: PCB bottom side

4 Discussion

4.1 Test results

4.1.1 Node and the Gateway

The results concerning the data from the system, shows great development of a low power IoT sensor system. With a low latency between the node and the localhost webserver. The system responded fast.
The battery life of the node which was set as a goal of 10 years was fulfilled and proven during testing with Energy trace, for the node with only the reed sensor capabilities. The testing of the node with light, ADC and reed sensor capabilities is not valid as the code contains a bug. The test was done as a guideline in the future concerning multiple sensor capabilities and it does show good results. The team thinks that in practice the node will not get an ack- packet from the gateway every time but neither 50% of the time, giving the reed switch only sensor node between 9.9 to 20 years of lifespan.
The range and signal integrity of the custom node, neither of which were tested as the team had no access to lab equipment. And even if the team had access there was no finished custom PCB to test the firmware on. The team had set in place different plans for testing and measuring radio characteristics. Ideally lab equipment to measure different radio characteristics would have given the custom PCB values that could be used in estimating the range and signal integrity. Something that was the most important reason for choosing a sub 1 GHz radio; however, NTNU put in place certain restrictions on campus where the lab equipment was located, so that even if the custom PCB was realized, the team would have to apply for employees of NTNU to do the measurements and tests. If there was no access to real lab equipment estimation of packet error rate could have been done, using programs like Texas instruments SMARTRF-Studio 7[40]. Something that would help further development in coding the node and the gateway. The current system using the TI LAUNCHXL-CC1312R launchpads as the node and the gateway had been tested for a small home with packets of sensor data reaching the gateway every time. The radio characteristics for the LAUNCHXL-CC1312R is seen below in Figure 42 and Figure 43.

Parameter	Condition	Typical	Unit
Sensitivity	50 kbps, 2 GFSK, DEV = 25 KHz, CHF = 98 KHz	-110.3	dBm
	SimpleLink Long Range Mode: 2.5 KBPS, 20 KSPS, 2-GFSK, Dev=5KHz, CHF = 38 KHz	-120.7	dBm
Saturation	Maximum input power level for 1% BER	+10	dBm
Blocking and Selectivity (1)	Wanted signal 3dB above sensitivity level , 50 kbps, 2-GFSK, DEV= 25 KHz, CHF = 87 kHz		dB
	±200 KHz from wanted signal	35, 34 (2)	
	±400 KHz from wanted signal	49, 47 (2)	
	±1 MHz from wanted signal	57, 56 (2)	
	±2 MHz from wanted signal	59, 58 (2)	
	±5 MHz from wanted signal	63, 63 (2)	
	±10 MHz from wanted signal	64, 64 (2)	
Rx Spurious emission	Conducted – 30 MHz to 13 GHz	<-98	dBm
	Radiated emissions measured according to ETSI EN 300 220		
	30 MHz to 1 GHz	<-57	
	1 GHz to 13 GHz	<-47	
RSSI dynamic range	Receive continuous mode.	95	dB
	Linear range from -115 dBm to -20 dBm		
RSSI Accuracy	Receive continuous mode.	±2	dB
	RSSI Offset = 0 dB		

Figure 42: 868MHz Receiver

Parameter	Condition	Typical	Unit
Max output power	At +14 dBm setting	14.2	dBm
Tx output power	At +10dBm setting (Customers need to Limit their Tx Power setting to +10dBm to meet Maximum Power ERP defined by ETSI EN300 220-2 Standard)	10.1	dBm
Harmonic emission	At +14 dBm Setting		dBm
	Conducted 2nd harmonic	-48.5	
	Conducted 3nd harmonic	-52.5	
	Conducted 4th harmonic	-58.2	
	At +10 dBm setting		
	Conducted 2nd harmonic	-43.6	
	Conducted 3nd harmonic	-59.6	
	Conducted 4th harmonic	<-59.6	
Spurious emissions	At +14 dBm setting		dBm
	Conducted in ETSI Restricted bands		
	Conducted below 1 GHz	-49	
	Conducted above 1 GHz	-56	
	Radiated in ETSI Restricted bands		
	Radiated below 1 GHz	<-57	
	Radiated above 1 GHz	-54	

Figure 43: 868MHz Transmitter

These tables were taken from the TI document SimpleLink™ CC1312R LaunchPad™ for 868 MHz/915 MHz Bands LAUNCHXL-CC1312R1[41].

Using a Microsoft excel spreadsheet from Texas instruments[42] the team estimated the maximum distance and link budget with the LAUNCHXL-CC1312R development board, Figure 44.

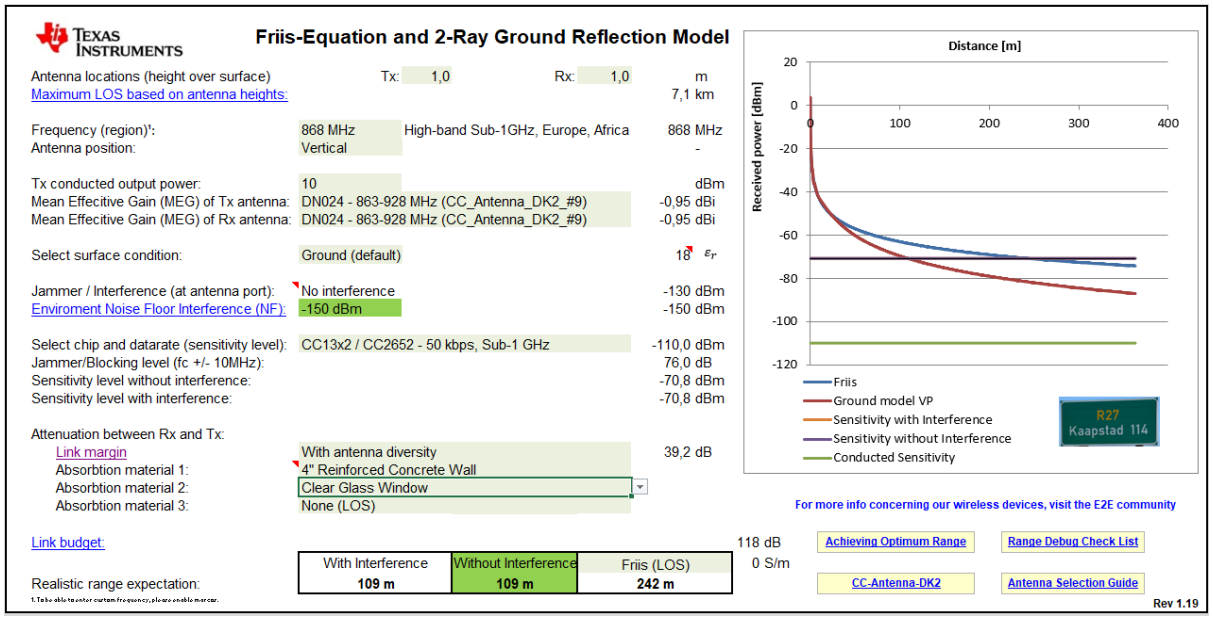


Figure 44: 4"/0.1014m reinforced concrete and clear glass

The link budget is 39.2 dB and realistic range is 109m in radius of the receiver; however, if there is no absorption material and a bit of interference(10dBm), the realistic range is 159 m. The link budget is then, 10dB. A typical wireless router in an indoor point-to-multipoint arrangement using 802.11n and a stock antenna might have a range of 50 metres. This of course depends on the frequency and transmission power; however, doubling the range for lower power is a definitive win for Sub 1 GHz.

4.1.2 Database and backend

The database results showcase the allowed and denied connections attempted to the database, as well as the sensor data stored in the database over an hour of testing. The tests revealed that all connections attempted were allowed, with no errors and no denies. This result indicates that all operations carried out by the uplink and downlink were permitted. None permitted results would be issues such as bad authentication or miss formatting.

This analysis tool was used thoroughly during development to ensure proper connections, it also displays how the security rules works in conjunction with the hardware and software developed. Several denies and errors was found during development, with one of the challenges being authentication from the uplink to firebase. This easy to use authentication is one of the advantages of Firebase. However, the features are matched, and on certain areas outclassed by competitors as detailed in 4.2.4 Comparison of database solutions.

The data displayed in the database were compared to the values transmitted. The results under KontoretTilArne and ntnuGateway08 were satisfactory, and like the transmitted values. The delay from values were transmitted from the Uplink until they were available in Firebase were estimated to roughly a second. These tests were highly satisfactory as a real time system was desired.

The frontend acted as expected and correctly displayed the same values as seen on Firebase. While simple in design it was found to be accurate, fast, and lightweight (it loaded instantly in Chrome).

4.2 Discussion of database, backend and frontend.

4.2.1 Initial thoughts

One of the weaknesses within the system is the open read privileges, as explored during method 2.5.3 Design. While these privileges are not “truly” open, as it requires the key, address, auth token and more, it does remain potentially breachable. If a hacker can retrieve or forge the key file, then the system is open to read at will. One way of bypassing this flaw is introducing a second manual authentication such as done on the uplink (ESP32). This would have to be integrated into the backend, namely fireBaseLink.js, or inside a new JavaScript file.

One way of achieving this integration is with the firebase module’s auth sub section. The framework is already established, however requires further coding work to implement user authentication. The user authentication allows users to be created and to be given various degrees of privileges. Accounts such as developer, admin and customer could be created with varying degrees of privileges. While this is not necessary for a simple IoT proof of concept system, it would be necessary for a final commercial rollout. This could be made in conjunction with a login portal using the Google authentication service. The Google Authentication service allows for users to use their Google users for ease of access. The system administrator could then manually grade the degree of access for every user or create an automatic system. The automatic system could for instance allow the customer to register with their google ID, fill in a secret verification code and have automatically assigned their IoT system to their privileges. So that they would only receive and be able to read the data for their own personal sensor system, which would be pre-defined by the developer.

Adding these additional security layers could gradually bring the system from a proof of concept into a commercially viable secure system. The availability of expansions should be exploited as it one of Node.js’s strengths. The usage of expansions allows for rapid and community driven development, as open source modules are intertwined with the team’s own code. There are several expansions that could offer interesting functionality for future iterations of the IoT smart system:

NodePlotLib allows for plots to be created directly within Node.js without any front-end preparations[43]. The module is free, open source and may be easily installed with npm. The package takes up 3.35mb and is inspired by the popular python package matplotlib [44]. The module provides three easy to use functions which are used to create plots. *Plot(), stack()* and *clear().* These plots could be used to graph the average sensor values measured, such as the temperature during a day. It could also be used to measure the average air quality in an office space. If the IoT system was connected to the air conditioning system, then it could adjust the air flow to ensure good air quality.

Another concern for future improvement is the security and vulnerabilities within the code itself. One way of working with these issues is directly manipulating and changing the logic of the code. One way of working with this security layer is adding IF and else Statements to strictly determine which processed may take place, and who may access them. These IF's would take conditions such as authentication, former attempts to establish connection and history into consideration. For instance, a new client that is asking for a lot of data, without having the proper authentication could be manually blocked if they failed to authenticate for instance 3 times. This could reduce spam, fishing and brute force attacks and best case prevent malicious usage of the system.

Another way of using logic is restricting what kind of data (length and type) may be accepted from Firebase. It could take the expected datatype and length and match it with the received data from Firebase. If the conditions for expected data were not met, then a warning or error could be prompted, and the system would perform the necessary steps to solve the situation or print an error log to the developer. This could for instance be the expectations for Battery Power of a sensor node. Where the value ranges [0,100], if the values are either a string or exceeding this interval, then the data could be rejected and not forwarded to the end client. The datatypes should be checked against Boolean logic because predetermining datatypes should not be performed in JavaScript, as it is automatically determined. This means that *VAR* for instance dynamically adapts to the necessary datatype to hold values. However, the content of the variable/object could be manually checked with Boolean logic. Functions such as *typeof* or *instanceof* could be checked against for instance '*string'* and *String,* respectively.

Extensions may be used in conjunction with the Boolean logic to supplement and form several additional security layers without interrupting the already established security with the Google Auth system. Retire is a command line scanner looking for known vulnerabilities in node projects. It may be used to scan the code for vulnerabilities. Helmet is an extension that helps secure the express application by manipulating and setting HTTP headers. It is best implemented in the middleware stack to ensure proper header definition. It can secure against potential problems such as *noSniff* (further explored in 4.2.8 Connecting to firebase) and *featurePolicy* (which may limit the features a front end site is allowed to perform). Lastly a handy extension for debugging and error handling is joi, which extends the possibilities and scope of error handling in Node.js.

One future iteration that might help value innovation upon IoT solutions is the usage of predictive AI. While not being within the recommended application of Node, Google offers a solution to run in conjunction and supplement Node. Google predictions has recently (as per 23th May) launched after it's beta testing stage[45]. Firebase Predictions applies Google machine learning to app analytics and allows the user to create segments on predicted behaviour. In future iterations of IoT systems this might allow the developer to find trends and predict future behaviour of systems and future events that are yet to take place. It will thus directly aid the user with more and potentially more accurate readings of their system environment (e.g. offices, homes and factories). It may also be used to alert the user if data trends suggest that fatal real-world events or dangerous situations are to take place.

The frontend has perhaps even more improvements that could be made, including the design (as mentioned with more graphical input) and functionality. New additions could be created such as session login (secured with google authentication as detailed earlier), more details about the application and proper CSS utilization. These features are strictly not necessary for prototyping and development, however, are expected for a commercially viable product. Lastly smart device support could be added with simplified view for small screens and touch.

4.2.2. WebSocket vs REST

In designing a backend system, it is crucial to make conscious decisions on where to use REST principles and where to use WebSockets. Although fundamentally different the two systems have their areas of brilliance and areas of lacklustre performance. While they may

perform similar tasks, their intended areas of application do somewhat differ. While both can sustain dataflows in large magnitudes, using one over the other could have direct consequences on the system. Furthermore, they are also different to set up, and operate with a vast array of variations on execution. Meaning that there are a lot of different ways to use them (e.g. different WebSockets available) and solutions available. This thesis has tested several solutions and has concluded that for the given IoT application a hybrid and combination of REST and socket.io yielded a good potential. Socket.io being similar, but not identical to a conventional WebSocket. The matter is further discussed in 4.2.3 Socket.io vs WebSocket.

The REST endpoint of the Firebase API served as an excellent interface towards microcontrollers and low power solutions. HTTP requests and responses are lightweight and easy to handle. They are natively supported by Espressif's ESP32 , and the <FirebaseESP32.h> library makes it easy and intuitive to work with and is highly compatible with the logic of C programming. An alternative future solution could be incorporating a rest endpoint inside the Node.js host. This was prototyped and tested using the Postman application. While the REST endpoint and successfully created (using Expressjs) it was beyond the scope of the thesis, and ultimately terminated. One advantage of setting up a proxy or alternative REST endpoint is the control it brings, as all data could or would have to go through the Node.js created REST endpoint. This may incorporate new security layers, filtering or other features that is not possible to create on the Firebase API's REST endpoint.

On the other hand, socket.io proved potent for interfacing between the server and the client for the backend. It dials up a connection and stays connected until the client is disconnected, meaning it does not have to redial and connect for every data that should be emitted. It does however require more modules, frameworks and adds increased complexity at the premium of real time connectivity. While socket.io could have been used for the microcontrollers (i.e ESP32), it would have been too extensive and unnecessary for the scope of the operation. Introducing additional data size and stress as well as complexity of setup for the backend. This is because supporting a larger number of sockets would require more capacity and a direct connection to the backend before interacting with Firebase, ultimately removing the need of Firebase completely. This is because Firebase does not natively support socket.io and could not interface with it. However, one advantage would have been the added security layers with encryption through socket.io.

4.2.3 Socket.io vs WebSocket

It is debatable whether socket.io is a de facto WebSocket or a competitor. While socket.io and a WebSocket in practice covers much of the same applications, there are some minor differences between them. Although seemingly more of a discussion of definitions, there are some differences in features and implications. One of the driving factors in the comparisons between WebSockets and socket.io is the popularity of the module. Socket.io has more than three million downloads weekly as per 27th of May 2020 [46]with a free reuse MIT license[47]. The project is community driven and gets frequent updates.

A custom module named *websocket-vs-socket.io* attempts to compare the data usage from socket.io and a standard WebSocket using Express.js (which is the exact same module as this thesis). While it is not representative for all cases of WebSockets and the sample base is small. It was taken into consideration while deciding between the WebSocket and socket.io. The creator Rafał Pocztarski carried out several tests and made the following observations:

- WebSocket created 2 requests at 1.50KB during 0.05s.
- Socket.io created 6 requests at 181.56 KB during 0.25s

These results indicate that socket.io does in this scenario (using expressjs) on front and backend does produce a lot more network traffic; however, the magnitude is still minor compared to the internet speeds commercialized today. Secondly the socket spends four extra steps in polling AJAX requests where the WebSocket does not. Lastly Socket.io consumes additionally 0.2 seconds extra, which introduces additional time latency. Although these tests speak in the favor of WebSocket, it is a small sample base, tested at a specific setting and conducted on localhost. Real world application may yield entirely different results. Secondly the difference remains insignificant for the scope of this bachelor thesis. Where the 0.25s delay seems accurate compared to the findings of the IoT systems similar application of socket.io. All in all, it poses an interesting question whether socket.io or a standard WebSocket should be applied. In future development testing other WebSocket's could yield connectivity and network usage gains, however, would most likely only be noticeable when undergoing larger loads. Other aspects such as encryption, safety, reliability, and implementation must also be taken into consideration.

4.2.4 Comparison of database solutions

Some competitors to the solution provided by Google are MongoDB, Redis and DynamoDB (Amazon). While DynamoDB provides a fully managed NoSQL solution such as firebase, a more interesting competitor to examine might be MongoDB, as it differentiates more to Firebase than DynamoDB does. It might appear that, at the publishing of this thesis, DynamoDB and Firebase aims towards the same "brand" oriented inclusions, where their offerings are aimed towards offering a Real-time database to Amazon or Google based systems. On the other hand, MongoDB appears to be more of a "standalone" application aimed towards any system, not pushing the developer towards more of its own products (such as Amazon or Google hosting).

One advantage of the MongoDB offer is the open source free of charge solutions. The only framework the developer needs to follow is terms as per the SSPL (v.1.) and Apache (v.2) license. The licenses stipulate that the developer may use their software solutions for commercial usage of any purpose. The only case where the developer will have to pay a fee is if it does not comply with the AGPL terms, and thus wishes to expand upon the licenses provided by MongoDB. In essence this indicates that MongoDB is a great solution for resource strong IoT start-ups with sufficient knowledge and man hours but lacks funding. MongoDB requires to be set up from scratch for backend usage with external hosting; however, it comes at the advantage of being free of charge otherwise, thus being excellent for university and educational usage. It is also widely supported, tested and transparent with vast documentation. The supported server operating systems include Windows, Linux, Solaris and OS X. This hosting solution needs to be set up by the developers, alternatively a 3rd party may be hired to set it up.

The programming expertise within a development team and the scope of the operation should be taken into consideration when choosing a platform. Firebase supports C++ Java, JavaScript, Node.js, Objective-C, PHP and Swift. With MongoDB supporting C, C#, Java, JavaScript, Perl, PHP and Python. While workarounds and clever libraries might be used to circumvent these supported languages, starting with one might prove faster development times for prototyping. MongoDB is the only solution which offers support for Python and PHP, while Firebase on the other hand supports C++ and Swift as well. Should iOS apps be developed without a middleman server, then firebase might be preferred. On the other hand,

being able to program in Python is quite handy for a lot of developers as it is often regarded as an entry level language.

Where firebase is somewhat straight forward in security rules and setup, MongoDB has more flexibility and complexity. One of the noticeable features is the LDAP authentication[48]. LDAP builds upon the pre-existing security measures which are included by default. Features such as password complexity and age-based rotation are not included on a fresh install. These features are suggested by several standards, such as PCI DSS[49] (Payment Card Industry Data Security Standard. These standards should be met if the scope of the operation is a commercial platform.

Unfortunately, the support of LDAP is only available for the Enterprise edition, meaning the community version may not suffice. Some workarounds are open source variants of MongoDB, which may be used in the enterprise versions place. Some key features included in Firebase missing from MongoDB are: Ensured SSL Certificates with encryption and dynamic smart configuration of ports. Some aspects that are included in MongoDB being harder to achieve in firebase includes: Password rotations and restricting connection to known network devices (Manually through auth).

MongoDB has a competitive feature set for a fully launched product compared to Firebase Real-time database; however, the disadvantage is the marginally higher setup and development time. It is thereby perhaps not recommended for prototyping with no pre-existing knowledge or while under time restraints. The Google catalogue provides an easier to set up pre-deployed database (i.e. “plug and play”), with support for the entire firebase catalogue (such as Auth, cloud hosting, diagnostics, data predictions). On the other hand, for enterprise and large-scale operation, MongoDB appears to offer a vast set of tools far superior to what a “vanilla” firebase console may offer.

One way of circumventing the lack of added versatility within the firebase console is adding a middleman server between firebase and the clients. A popular open source solution being Node.js, which is also well documented and supported by Google. If firebase rules are used in conjunction with Node.js, Google Auth and a smart Node.js application, then the given application will be able to compete well with MongoDB’s upper hand in security and flexibility. One major advantage MongoDB has, is the control over hosting should premium

hardware be utilized, as the developer may have full control over the hosting process. While Firebase is hosted by Google and may not be customized for the average user.

For the given IoT smart application the superior choice appears to be Firebase. It brings an easy to use platform already including the google firebase catalogue. Including a login gate, data storage, analytics and a real time true and tested platform with high uptimes. Firebase is also a user-friendly platform aimed to aid start-ups at early development stages. It features well documented code examples showing important setup steps with the Google level of polish. Lastly firebase requires no external hosting and may be used after merely minutes of setup. Given these crucial advantages, Firebase becomes the platform of choice for this bachelor thesis. Should further development take place beyond the proof-of-concept scope, then MongoDB should be further explored as a main database solution. Because it offers a free of charge hosting solution. The only challenge is establishing reliable hosting, which in itself might introduce extra cost and defeat the purpose of the transition. Another advantage is that MongoDB might enhance the performance as it is a dedicated hosting solution, which provides more flexible and local data hosting. The database might be internal to the developer's offices or in close proximity. This offers shorter physical data flow travel paths, and may thus serve data more securely, efficiently and faster. Following a known up- and downlink, such as a LAN solution (if feasible to the scope of the project). If handling critical data, a local database may be set up to prevent connections from forming outside of the local network. These LAN solutions are possible to create with MongoDB as it is a standalone solution.

4.2.5 Node.js

While Node.JS has several areas of usage, it will exclusively be used to host the web solution and backend server, which can be seen as a "middleman" server. Alternatives to using Node.JS in this formation are other systems such as Ruby on Rails (Web-application framework). For the given application Node.js will not act as the REST endpoint or database. Node will act as a buffer between the applications (i.e. web browser) and database (i.e. firebase). This formation introduces a secure barrier between the nodes in the network and the client. The barrier makes sure that the uplink (i.e. ESP-32) will never be directly connected with the web client. So that in case of attacks, or malicious usage of the web application a last layer of defence is still present. All data flow must always go through the

middleman server, in this case being Node. While this does not directly prevent attacks on the database, nodes or web application, it does prevent abuse within the provided data link itself. The middleman server can thus be seen as a "filter" between the database and the client application. Answering requests and providing responses should the query criteria and authentication meet the requirements.

This logic may be built either with Boolean logic (such as IF, ELSE) and or in conjunction with modules built into Firebase and other first party extensions. Node will also act as an asynchronous state machine, dictating whether the data that is requested has been forwarded from the database, and is ready to send to the client. Or in the other case where data is yet not ready to be displayed, and that the client has to await data confirmation. This asynchronous callback structure is one of the strengths and intended usage of Node. The decision of using Node instead of Ruby On Rails and other PHP driven architectures was taken on the back of these features and modules, which may easily be implemented in Node. With the simple "NPM MODULENAME" providing an ocean of open source modules as well as the JavaScript based programming. This workflow and support by companies such as Google (with their firebase module) adds industry standard functionalities at no cost with an easy to implement workflow. It also allows local prototyping and testing at no expense. All these factors combined made Node.js the software of choice for middleman server (backend). Following in the footsteps of companies such as PayPal, performing similar transitions[50].

While not being a programming language in its own rights (being a JavaScript runtime environment for the w8 engine, based on C++). It is an asynchronous (non-blocking) structure which allows the system to handle several requests at once (compared to ASP.Net and rails which works synchronously (standard without modifications)). This choice of software solution frees up resources as the system will not wait for one task to be performed at a time; however, it is not recommended to use Node for CPU intensive tasks, such as image analysis etc. Because of this node should be fit in application as a middleman server. Should future iterations of the systems require more CPU intensive tasks, then a secondary server might be recommended to handle these processing needs. These operations might include heavy data processing, image recognition and artificial intelligence/machine learning.

The design philosophy behind the programming is a modularized system. A modular system is preferable both because of the added flexibility, readability and reusability, but also for

ease of execution and debugging. Using the support for modular approach in Node, being the export function. It allows select functions to be defined in separate files and called in the main file (i.e. master/index/run file). “module.export.NAME.” includes the given functions and variables from the modules and include them in the master file; however, only the needed functions, objects, literals, variables or expressions are exported. Everything that is not required in the master file will remain exclusive to the modules, such as sub-calculations, nested functions leading to the necessary functions, conversions and so forth. This modularity, ease of use, readability and efficiency is one of the advantages with the node.js design. This modular ensures a future proof system, where several engineers can work simultaneously independently on various packages which make up the entire system.

Following good readability and coding precedence, only the required functions and variables will be exported. Having the functions and variables, that makes up the main function be local to the module. One area of application is the authentication file that holds the keys, addresses and other important variables required to establish a legal connection with firebase. This file is thus kept outside the main file, increasing code readability and reusability. Should the same main file be used with another firebase, then a new authentication file may simply be included in its place. Allowing one line of code to indicate a new connection, rather than directly including all of its content. This may also dynamically be used to include and or change connections within the operation of the backend application. This IoT application seeks to build a framework which supports ambitious future iterations and design goals. It follows the intention of providing the best possible framework, while still holding true to the “proof of concept” ambition. Seeking to find a balance between ambitious and realizable. Variables and functions will thereby be exported on a need only basis to the main module (the master file).

Global variables are avoided in the server backend and the programming, where possible. Variables and functions will therefore be kept locally to all files, unless exported. While the usage of global variables is possible to make interaction between the modules easier, having global variables may also sabotage and provide unintended system bugs and longer naming conventions. With regards to the scope and complexity of the IoT system, having global variables would most likely be achievable without indicating uncaught errors and long variable names; however, it would have to be improved or changed in future iterations, which goes against the overall design philosophy. Seeing how custom solutions are often needed for

smart systems for various usages spanning over several usage areas. Such as fabrics, homes, universities, automotive industries etc., with vastly different needs and requirements. Thus, having a stable core with expansion ability and minimal overlap is desired.

Following the recommendations by the Node.js documentation, usage of synchronous functions was avoided where feasible. Following the design philosophy, the backend was designed as a non-blocking single threaded node (by default). In the case of an IoT online application a magnitude of clients will be connected simultaneously and there must be capacity supporting magnitudes similar to the scope of the system (e.g. a home, offices or a factory). In case of a single home, or a section of offices, a blocking approach might have been sufficient; however, in the case of a larger scale operation a blocking node structure will prove difficult to execute. The timing from firebase for a full GET is estimated to a roundtrip of 500ms - 1s under ideal circumstances. These numbers are also likely to increase under high loads or with limited connectivity. With longer wait times, and an increase in userbase having a blocking node single threaded structure will in worst case render the system useless. Utilizing the recommended non-blocking structure was deemed necessary for a future proof design and was implemented with highest priority.

The node asynchronous method takes a function as their last argument (alternatively Promises or Async wait). This will “callback” the function when the second function has successfully reached its end condition (e.g. return). The process may be compared to the somewhat similar interrupt driven C programming style. In order to achieve the asynchronous structure callback structures were used. Where events (e.g. functions) will wait in a logical order for the prerequisite to complete its task. The most important usage of callback functions is in conjunction with firebase, as previously mentioned with timing purposes. If similar operations were to be built using synchronous systems, then timing must have been fundamentally changed, and ordering must have been structured accordingly. Having long wait times and slow responses.

Events make up one of the core functionalities and strengths of Node.js, having much of the functionalities and guidelines built around this structure. Events are used as triggers for the program indicating at which state the program should be at. For every trigger there is a listener, which remains idle until the listener is called when an event is raised, such as a callback function. One of the main challenges with these approaches, and the asynchronous

workflow is planning and executing a logical event progression system. Where for every event it must be ensured that the overall desired task will follow the chain of events until reaching the desired state. The complexity of these chains requires a detailed diagram at which the development team must accurately implement in code. It must be ensured that there are no recurring loops slowing down or making a state stuck in repetition in the case of recursive feedback. In essence the progression must await the former state, and then feed the important data down the chain, creating an asynchronous data- and logic flow with low latencies and as little wait time as possible, for a seamless user experience.

The in-code structure that must be followed to ensure this logic and program flow is using the aforementioned callback functions. Node will automatically sort the logic and register the listeners that are called upon. Furthermore, all code is repeatedly called, and the logic will run infinitely if it is properly called. Another important aspect is the inclusion of "return" at the end of callback functions, where it returns data to be passed through the event chain; however, the return statement must come at the end of the function's logic regardless of which state is reached (i.e. so that code lines are not lost). So that several returns must in some cases be included if logic such as IF, ELSE is implemented (so that every scenario has an end condition or abort).

Another important aspect of asynchronous event construction is accurate and elaborate error handling. Building a network of efficient and descriptive error conditions, if the event chain has reached a dead end or a false state. This may prove a key aspect to prototyping and testing within Node (and most other backend systems). It will also help detect errors in real time application and early warnings for malicious usage of the system. These error conditions can be built for cases that are not intended to be reached, such as outside the scope or with unrealistic wait times. This may be achieved by timing events or creating system states that are not to be reached. While timing is handy during debugging, it will introduce a strain on the system if kept for end results in vast numbers (as the timers must also count). Although some timers are sustainable, too many might be counterproductive. While the usage of error statements and flags may be used in greater magnitudes without further repercussions on the system performance, as they are called once per error event rather than as a precaution.

4.2.6 Firebase Real-time Database Security Rules

Accessing data directly from a firebase without a middleman server is theoretically feasible; however, it might not be recommended in practice with regards to system integrity and security. While it is possible to access data directly from mobile devices bypassing an application server, in practice it might prove risky. It is documented by google that sufficient security and data validation is available for a direct connection. This security is ensured directly through firebase Real-time database security rules. The security rules allow the developer to define rules for retrieving and pushing data to the database. In practice having an application or backend server, serving as a middleman, is presumably in most cases still advised. Google offers a well-documented, open source and community driven solution for Node.js. Which in combination with a safe and well-defined security rule setup allows for secure data management.

The Firebase security rules follow a predefined syntax and structure created by Google. It is used to secure the data stored in Real-time Database, Cloud Firestone and Cloud Storage. The task of the security rules is to act as a barrier between the data and malicious users. The security rules are already built into the console and may be set up to the level of granularity desired for the given application and security measures needed. The firebase Real-time database rules leverage JSON in its definitions. On the other hand, Cloud Firestore and Cloud Storage leverages a unique and different language designed to accommodate a more complex rules-specific structure.

One major advantage of this approach having the rules defined directly into the console, is the distancing from the client applications. The client application may not under any circumstance change the firebase ruleset. The rules may alternatively also be changed in the Firebase CLI.

The logic supports OR logic but does not implement AND statements. One challenge with this approach is the occurrence of overlapping ruleset. Where OR allows several conditions to meet the requirements, and still hold true to Boolean logic. While AND could perhaps prove more efficient, as a development aid to counteract false logic it is recommended to pay attention to the security rules flag overlap warning. The suggested implementation path of the security rules follows a four-step guideline:

1. Integrate product SDK (Set up real-time database for the given application).
2. Write Firebase Security Rules (set up basic rules)
3. Test the firebase security rules (Use real-time Database Emulator to validate the applications behaviour to validate rules before deployment.)
4. Deploy Firebase Rules

The advantage of following the given guidelines is the security at which the application is handled. Being an always accessible application on the internet is dangerous, as several developers have met hefty costs with unfortunate or deliberate sabotage and exploits of unsafe security rules[51]. One absolute minimum and requirement while developing an application and IoT system is to turn off read and write while testing is complete. Read and write privileges were always turned off during the development of the Low Power IoT smart system where connectivity and testing was not being carried through. This is a secure measure that should take place while clients are not connected to the console, and the security measures are still not yet fulfilled to an appropriate level of granularity.

4.2.7 Comparison of Cloud Firestore and Real-time Database

The main difference comes in the ambition of the scope of the IoT application, and the role that firebase is to take in a system chain. While firebase must store data, how much advanced querying, sorting and transaction it is to perform is a key factor in deciding between the two solutions. Cloud Firestore has the richer and faster query system which scales well beyond the scope of Real-time Database. Another factor in the decision between the consoles is the magnitude of data flow throughout the system. Google recommends a scope of a few GBs or less of data with frequent changes for Real-time Database. While Cloud Firestore shines in high density situations of hundreds or thousands of GBs which are read more than changed.

Another advantage Cloud Firestore holds is an edge up on Real-time Database when connectivity is limited or missing, where it has wider support for continuous querying under difficult circumstances. On the contrary Real-time Database applies a function named “Presence”, which allows the database to record client connection status. It will then provide updates every time a client changes its connectivity status. While it is possible to implement such a function into Firestore, as documented by Google, it has to be manually set up and configured by the application team. Real-time Database supports upwards of 200 000

concurrent connections and 1,000 write/second in a single database with no limits to writing rates to individual pieces of data. It is possible to scale beyond these numbers if required by sharing data across multiple databases. Cloud Firestore supports five times the concurrent connections and ten times the writes/second. A drawback with Cloud Firestore is the limits on write rates to individual documents or indexes contrary to Real-time Database to its JSON tree.

Another major difference between the platforms, and perhaps more important to the coding department of a developer team, is the data model. The data of Real-time Firebase is structured in a simple JSON tree, following the JavaScript Object Notation (Especially handy for Node.js and Web Applications). On the other hand, Cloud Firestore offers a more complex document organized collection system (still similar to JSON). Another interesting side note is the guarantee of uptime by Google. While Cloud Firestore is guaranteed to have an uptime of 99.999%, real-time database has a slightly lower guaranteed uptime of 99.95%. If having a multiple database under console and account is preferred, then Real-time Database has to be chosen. It has multi database support seamlessly integrated into its SDK which may be accessed in the CLI (e.g. MS Powershell). With this setup several databases may be attributed to each customer and monitored within one Administrator user. Thus, allowing an easier workflow for separating databases into smaller sections, while still maintaining a competitive and wholesome overview of the situation.

Cloud Firestore has some advantages with its security rules and implementation. Real-time Database has only support for Real-time Database Rules. With the option of validating data separately using a validation rule, these rules must be manually set up and may dictate the privileges of data contributors. In essence dictating whether a single contributor may or may not write to the database. Cloud Firestore has heftier non-cascading rules with integrated authentication and validation. It integrates similar security rules to mobile SDKs as does Real-time Databases; however, it integrates a different Identity and Access Management to server SDKs. Furthermore, rules do not cascade unless a wildcard is used. This is a significant security measure to data security. A cascading rule dictates that any child of the affected parent, also will be manipulated by the same given status. This way greater flexibility may be attributed to the rule process, as rules do not cascade unless specified (i.e. by a wildcard). Rules may by this method be highly specific without introducing ripple effects or unintended consequences.

Cloud Firestore and Real-time Database both offers the same base functionality, however it is apparent that Cloud Firestore and Real-time Database are aimed at two different markets. Cloud Firestore offers a complex and highly dynamic secure solution with more finesse than Real-time Database. It allows the developer to manually adapt and fine tune every single operation with a fine comb. On the other hand, Real-time Database offers a simpler solution, which is more hands on and appears quicker to get going out of the box. It also has a more straightforward, IoT friendly pricing model and database structure. Allowing one user to control a vast array of databases from within its administrator CLI. While the query language and potential are somewhat reduced, it still performs the same basic functions, and still has a quite hefty support for data organization and fine searches. Data may be requested by key, children and organized by any preferred method, or simply by the root of the JSON tree. Having a simple data structure such as the JSON tree will prove sufficient for smaller scale operations, such as storing node information and sensor data. A logical setup would be having the parent be the identity of the data that is to be stored, and having the children named after the information that is contained within. Meaning the parent is the name of the data, and the children each hold the acquired data under another name. An example of this may be:

```
  },
        "ntnuGateway08" : {
              "batteryPower" : 100,
              "doorState" : 1,
              "temp" : 27
  },
```

Where NTNU Gateway08 is the nodeID, and the children display their name and value. Such as batteryPower: 100. This is a clean way of reading data for a human operator, but also more importantly widely supported by the recipients and clients that are to use the data. These data are stored in the firebase from the finished IoT low power solutions, as detailed further in this thesis.

Another important difference are the security rules presented by the console, while different and definitely more potent in the Cloud Firestore. If used properly and in conjunction with a

middleman server; then both solutions are expected to offer a secure system that may meet security standards, such as PCI DSS. While IoT applications do not always function within the eCommerce sector, following the guidelines such as PCI DSS is a good starting point for a secure system as they apply to any given application containing users and delicate data. It is important once again to remember the delicacy of IoT data, as it collects data from our homes, workplaces and offices. This data is highly attractive to third parties which may wish to exploit and or profit off the data stream. This thesis, while still in the proof of concept stages, has spent a great deal of time investigating and theorizing security solutions. It is the aim of this thesis, to investigate and suggest the important security measures that have to be taken in an end stage system implementation (as suggested and stressed during the length of this article). As seen in the simplified link diagram

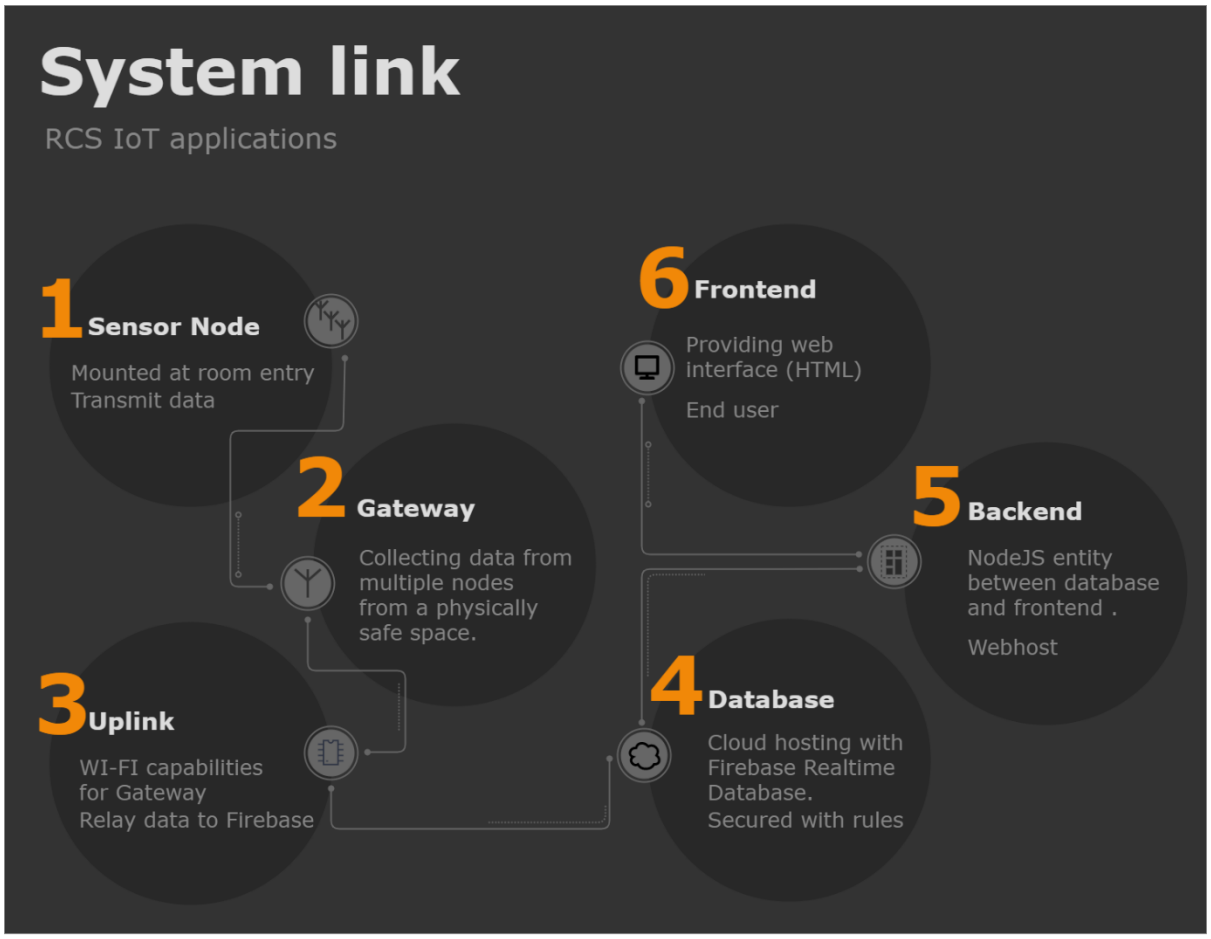


Figure 6: Full system architecture Figure 6: Full system architecture. The usage of a middleman server effectively removes the direct connectivity from the client application to the backend database. Meaning with the given setup two security layers are presented between the request and the response, and vice versa. While not fully exploited, nor implemented the system theory allows for a futureproof modular system that allows security

measures to be a first priority, without excessive system modifications beyond the scope of their intentions.

There is also a pricing difference that matters highly for IoT systems, with Real-time Database only chagrin per bandwidth and data store. In the current build of the prototype system, only the necessary data is uploaded and downloaded from the system, at change of data. Meaning the system will only receive data from the link at data received throughout the low power chain. And it will only be read when the middleman server is refreshed and forwarding data to the web browser. Because of the small magnitude of the dataflow, only a small price has to be paid for the data in the system. At the time of 20th of May, in the final month of development. A mere 1.5KB is being stored in the database, with 6.7MB of database downloads, coming at a peak of 5% load from a continuous stress testing through the ESP32's (uplink) Wi-Fi modules upstream connection. This suggests that at the free data plan, with the current low power low data density configuration. 20 of the prototyping solutions uplinks (Master Node) may simultaneously transmit data to the real-time database. This scenario is highly unlikely to occur as the data transmission time is very low in the low power application. Data is only being sent from the CC1312 node at slow intervals to retain battery capacity, meaning the data intensity throughout the chain is very low. These early measurements, while not conclusive. Suggests that with Real-time Database on a single database several hundred uplinks may be connected (given random intervals that do not overlap more than 20 writes momentarily). Seeing how a RESTful API is to be utilized (as detailed by this thesis), and connections are not made through persistent WebSockets. It is realistic to assume that system integrity will be upheld, and that the model of choice will be secure, cheap and reliable for the given IoT application and the scope of this operation.

Lastly individual database support within a single system is a good security measure to which should be deployed if the system is to be used by various sites and real-world systems. Given the following scenario: The IoT low power smart sensor system is to be deployed at NTNU Gløshaugen and NTNU Kalvskinnet. These are two systems that are to act independently and will report to two independent subsystems. These systems must be separated for security measures, as data regarding these campuses are highly confidential. Should a data breach be detected, then the one system must be able to be taken offline while the other one should be able to act independently. In this case having one database will prove difficult and also reckless in the face of data security.

Should the two campuses also desire custom ruleset as a result of custom requests, then the added flexibility of two separate databases will prove rewarding. Also, in the case of a hostile takedown, data breach or attacks then having two distinct separate instances may keep the other from going down. These two databases will have a different connection key, ID, authentication and relative URL while still being manageable by the same Admin system (Heavily Secured). This hard barrier between interconnection is key to a secure IoT system. Another added benefit is the differentiation of the databases, meaning twice the read write cycles will be available evenly distributed between the two databases. Meaning given our early measures, that 20 uplinks may be connected momentarily to each of the discrete subsystems at any given time. In this scenario, choosing Real-time Database over Cloud Firestore will both provide a smart and easy solution to separating the databases for ease of use, increased bandwidth, capacity, connectivity and support.

The two disadvantages in the given scenario is the loss of advanced security rules (at the gain of a database separation). As well as the loss of advanced querying and the advanced bad connectivity aid implemented by Cloud Firestore. Based on the scope of the operation, and the parameters desired by the application, implementing Real-time Database appears to be the superior decision. While Cloud Firestore will be continuously updated, and at a later stage may have implemented the missing features, as it is Google's real-time database enterprise solution and flagship.

The downsides of Cloud Firestore may also be circumvented by smart usage of middleman servers such as Node.js and its vast library of modules. Which will probably in combination bypass the potency of the similar combination of Real-time Database and Node.js combined. However, said application will require a greatly larger workforce, and man hours, making it unrealistic and far too ambitious for a Bachelor thesis, and will not be touched upon by this thesis. Further mentions of Firebase in this thesis will strictly relate to Firebase Real-time Database and not Cloud Firestore.

4.2.8 Connecting to firebase

There are various ways of connecting to firebase within the scope of this thesis. One way of connecting to the database is interacting with it as a REST endpoint, with communication following the HTTPS protocol. Firebase will then only respond to encrypted traffic and will not respond to HTTP requests without advanced encryption, thus HTTPS. Secondarily another alternative is initializing the real-time database JavaScript SDK[52]. These SDKs may be accessed by including them in either the Browser (Optimal performance in chrome with the Google W8 JavaScript engine), or in a backend/middleman server (such as Node.js).

The second option rather than accessing the REST endpoint is a JavaScript solution with the supported SDK, as featured in the final NodeJS solution. This method follows a "higher level" of code w.r.t bot and top-level code. Meaning the developer does not have to handle the HTTPS requests manually, but through predefined functions and solutions provided by Google. This procedure, using the firebase functions as further detailed in the rundown of the final backend code, is better fit for the current needs and requirements of the backend. While the ESP solution on the other hand still relies directly on the basis, at which the REST endpoint is provided.

4.2.9 Testing with the Firebase CLI SDK

The admin CLI was strictly not necessary for Node.js implementation; however, it greatly reduces redundant code clarification and complexity. It was used during initial setup to test the firebase functions, reassure connectivity, and select the desired project and database to be used with Node.js. It is also a simple introductory step that allows the usage of the firebase CLI to investigate and debug firebase connections [53]
The following debug was produced during a testing session with the CLI SDK.The file confirms the connection, CLI version, Node version, date and how the user has authenticated to the desired database. As seen in the debug file, the user has prompted a retrieve of projects:list which is a firebase SDK specific command. The command issues a request to the firebase for the available project names. The database receives a HTTP GET request and responds with a HTTP response of 200 (ok) as well as a body with the desired Payload (i.e. list of projects).

```
[debug] [2020-05-15T09:47:48.713Z] ------------------------------------------------------------
--
[debug] [2020-05-15T09:47:48.715Z] Command:       C:\Program Files\nodejs\node.exe
C:\Users\kbhro\AppData\Roaming\npm\node_modules\firebase-tools\lib\bin\firebase.js
projects:list
[debug] [2020-05-15T09:47:48.715Z] CLI Version:   8.2.0
[debug] [2020-05-15T09:47:48.715Z] Platform:      win32
[debug] [2020-05-15T09:47:48.715Z] Node Version:  v12.14.1
[debug] [2020-05-15T09:47:48.716Z] Time:          Fri May 15 2020 11:47:48 GMT+0200
(GMT+02:00)
[debug] [2020-05-15T09:47:48.716Z] ------------------------------------------------------------
--
[debug] [2020-05-15T09:47:48.716Z]
[debug] [2020-05-15T09:47:48.725Z] > command requires scopes:
["email","openid","https://www.googleapis.com/auth/cloudplatformprojects.readonly","http
s://www.googleapis.com/auth/firebase","https://www.googleapis.com/auth/cloud-
platform"]
[debug] [2020-05-15T09:47:48.725Z] > authorizing via signed-in user
[debug] [2020-05-15T09:47:48.730Z] > refreshing access token with scopes:
["email","https://www.googleapis.com/auth/cloud-
platform","https://www.googleapis.com/auth/cloudplatformprojects.readonly","https://www
.googleapis.com/auth/firebase","openid"]
[debug] [2020-05-15T09:47:48.730Z] >>> HTTP REQUEST POST
https://www.googleapis.com/oauth2/v3/token
 <request body omitted>
[debug] [2020-05-15T09:47:49.159Z] <<< HTTP RESPONSE 200 {"content-
type":"application/json; charset=utf-8","vary":"X-Origin, Referer, Origin,Accept-
Encoding","date":"Fri, 15 May 2020 09:47:48 GMT","server":"scaffolding on
HTTPServer2","cache-control":"private","x-xss-protection":"0","x-frame-
options":"SAMEORIGIN","x-content-type-options":"nosniff","alt-svc":"h3-27=\":443\";
ma=2592000,h3-25=\":443\"; ma=2592000,h3-T050=\":443\"; ma=2592000,h3-
Q050=\":443\"; ma=2592000,h3-Q049=\":443\"; ma=2592000,h3-Q048=\":443\";
ma=2592000,h3-Q046=\":443\"; ma=2592000,h3-Q043=\":443\";
ma=2592000,quic=\":443\"; ma=2592000; v=\"46,43\"","accept-ranges":"none","transfer-
encoding":"chunked"}
[debug] [2020-05-15T09:47:49.167Z] >>> HTTP REQUEST GET
https://firebase.googleapis.com/v1beta1/projects?pageSize=1000
```

This debug file shows the conversation with the database as well as minor details such as dataformat (JSON), charset (UTF-9) amongst other crucial information. It allows the developer to prepare for the given formats and conclude whether the database was configured correctly. In this case it is confirmed that the database is configured correctly with a JSON datastructure as selected during setup, and further explained in 2.5.2 Data structure. It also provides insight into the details of the conversation between NodeJs and the firebase. Features such as *X-content-type-options: "nosniff"*, which simply put is prevents programs such as web browsers from performing MIME sniffs (also known as Content Sniffs).

Another feature which firebase introduces is the *x-xss-protection* which is set to 0. This feature is part of the HTTP protocol and stops pages from loading should cross site scripting attacks be detected. This may appear as a security downgrade as it is set to 0, which disables XSS filtering. However according to Mozilla (developer.Mozilla) these features are redundant in modern browsers, as a strong Content-Security-Policy will already cover the inline Javascript ('unsafe-inline'). Furthermore, it is clarified that it might be recommended should older web browsers without CSP (Content-Security-Policy) be utilized.

Having access to this information regarding the traffic between firebase and Node.JS is crucial for debugging and testing. Inspecting the communication between firebase and Node.JS reveals the care that is put into securing the communication therebetween and suggests which areas Google has already covered with regards to the security layers. Both web browser sniffing and XSS filtering is already addressed as part of the Google module. Another interesting aspect are the timings of the transaction, where the window was approximately one second. As the system become increasingly complex studying the traffic and details of the communication might prove important in securing the integrity and security of the system, having the .debug function as a step towards better insight.

4.3 Choosing the Uplink MCU

When the planning of the project started, it was agreed that it was preferable with a lightweight, cheap, small profile yet powerful microcontroller to handle the firebase-communication. The team also considered an integrated antenna and easily accessible UART-pins a huge advantage, as it would make our setup more convenient. The team has experiences with the Arduino Uno from earlier projects, but concluded that it was insufficient for multiple reasons, and compared multiple Wi-Fi-supported microcontrollers.

The Arduino Uno could possibly handle the workload on a scaled-down system, but its main drawback is its single core 16MHz, which will quickly throttle the system during large workloads, possibly crashing or posting the data with a huge delay. The team consider the lack of an integrated Wi-Fi-antenna a huge disadvantage, making the design of our system unnecessarily complex. These points made it an uninteresting choice for the uplink.

The TI CC3200, Realtek RTL8710 and Nufront NL6621 are all comparable to the ESP8266, however with some small deviations from the specs. However, each one also has its flaws not found in the ESP8266. The TI CC3200 has great support but is considered too expensive if the focus is a low-cost system. Both the RTL8710 and NL6621 cost more than the ESP8266, and considering the stock problems, it would have been too risky to base the project on getting hold of one of these.

After some consideration, and comparing the specifications with the project's needs, the team chose to use the ESP32 as the main Wi-Fi-microcontroller. Offloading the Wi-Fi-workload to one core, while the other can handle the data uninterrupted means that the system can more reliably remain stable during large workloads. The higher available core clock speed also means that it can be adjusted to handle an even bigger payload, which means that the ESP32 will realistically never throttle the system. The ESP32 also has a lot of documentation, libraries and functions shared with the ESP8266, which means it is easy to program, troubleshoot and generally work with, especially considering the number of external libraries and functions supported by the ESP32. The team also have the advantage of some basic prior knowledge about the ESP32, which makes it easier to set up the fundamental code pretty quick and have the option to go more in depth at a later opportunity.

4.4 ESP32 Programming

4.4.1 Communication protocol between CC1312R gateway and ESP32

There are multiple communication protocols viable for inter-module communication, the most common ones being UART, SPI, I^2C or wireless protocols such as WI-FI, Bluetooth , Zigbee etc. Wireless communications were however considered unnecessary, as the team based the uplink on a low-power node, and the wireless technologies supported by the ESP32 does not have the low-power versions, such as Bluetooth Low Energy.

UART is an asynchronous protocol, which eliminates the need for a common clock, solving the issue by start- and stop bits. It is however limited to a maximum data frame of 9 bits if no parity bit is used. The parity bit can however be enabled to allow error checking, but this limits the data frame to 8 bits. The start and stop bits reduce the complexity of a system, only demanding one pin on each end given that simplex or half-duplex is used, while I^2C utilizes minimum 2 and SPI needs 3 pins. The only prerequisite for UART to work on any system is a within +-10% matching baud rate and the same bit setup. UART is however limited to one-to-one communication, while I^2C and SPI supports multiple nodes.
UART is an asynchronous protocol, which eliminates the need for clock, solving the issue by start- and stop bits. It is however limited to a maximum data frame of 9 bits if no parity bit is used. The parity b

4.4.2 ESP-IDF

Although the team chose to use VSC in combination with PlatformIO, Espressif has its own development framework for programming the ESP32. It is however not a simple library that can be easily added to VSC, but rather a software for programming and uploading code, with its own included modules and toolchain.

One of the main advantages of ESP-IDF considering our system is more control over the UART settings and communication, making fine-tuning and better error-finding more prominent and intuitive, rather than the simpler methods used in the projects code for squishing out bugs available by using VSC and PlatformIO; However, these has given more than enough tools to make a working system without any big hindrances.

A drawback of using ESP-IDF is its rather cumbersome setup and learning curve. One must ensure proper setup of Python in the OS, download the correct packages, make sure to integrate it correctly via settings in VSC, and install the correct drivers. If one wants to rather use the ESP-IDF toolkit, there is an official guide online [54] and its own GitHub page[55], however it will not be covered in this report. There also exists an ESP-IDF plugin for VSC if one wants to avoid the recommended toolchain, at the risk of worse support and integration, and more a higher chance of bugs and incompatibilities.

If one were to use the standalone software for ESP-IDF, you might see that its user interface is quite more simplistic than VSC, more reminiscent of DOS 3.0 or some older BIOS-menus, which can be detrimental for some people. While one can become fluent in using it, the team consider it hindering in development speed of the project when not fluent in using it and consider this a huge drawback for our workflow.

Due to the amount of documentation needed to read to get started compared to the teams prior knowledge of C using Visual Studio Code in combination with PlatformIO, it was concluded that it was easier and more stable to base the uplink on systems that the team has prior knowledge in rather than delving into the ESP-IDF libraries, as to make sure the system was completed in time, giving the team more time to focus on more pressing matters. However, the ESP-IDF libraries are quite interesting, and the uplink code might be considered to emigrate to it in a future revision of the project.

4.4.3 Errors encountered in the uplink

4.4.3.1 UART miscommunication - UART lost bits problem.

A re-occurring error between the gateway and the uplink is the loss and/or skewering of the transmitted bit, most likely due to some kind of desynchronization or interference. The UART accepts the string until it reads a null terminator; however, there are times where it somehow reads a wrongly formatted string that somehow is correctly terminated. A standard correctly formatted string looks like “NodeID:value1:value2:value3:”. There are however instances where the NodeID is missing, where some values are missing, where only a value is transmitted, and where a regular string is appended to a faulty single transmitted value or vice

versa. It was an error that occurred early in the programming, but the teams troubleshooting has given no clear answer as to why this error persists, even after remodelling the code. Another problem most likely related to this, is missing letters from the string. During most of the testing phase, the “NodeID” was always set as 11 characters long. Multiple times however, the uplink received a NodeID that was >=10 or <=12, which caused spam to the firebase database by posting shortened NodeIDs.
The following picture is an example of such faulty code, where a random value is put before the NodeID

```
3:j5Ijs9kJ3:33:0:1:25:18:-16
```

Figure 45: Example string with error

4.4.3.2 Guru Meditation Error: Core 1 panic'ed (LoadProhibited). Exception was unhandled.

One of the most prominent and confusing errors that the team encountered during programming was a crash on the ESP32 most likely caused by incorrect parsing of the string. The expected string has a null terminator at the end, and by using the Arduino function

```
Serial2.readBytesUntil('\0', Temp, Index);
```

It reads the string until it finds the null terminator as normal; this is most likely caused by the fact that it omits the null terminator itself from the string, which results in a crash when the loop reaches

```
token = strtok(NULL, "\0");
```

Where it tries to split the string. Since the delimiter is missing, the default seems to be a hard failure, crashing the ESP32.
This problem persisted if any of the values are wrongly formatted by dropping some of the values, as proper error handling of such instances is yet to be implemented

4.4.4 Attempt at solving the Uplink code errors

4.4.4.1 UART miscommunication

As mentioned in “4.4.3.1 UART miscommunication - UART lost bits problem.”, multiple problematic strings plagued the system testing. To solve these issues, the team introduced a workaround which only accepts the string if the NodeID-part has the correct length after a parse and check, as explained in section “2.4.4 Operations of the Uplink”. After this workaround the string was no longer accepted if it did not have the same length as the pre-agreed length from the CC1312. This was a band-aid, but it no longer accepted non-fitting input by skipping them, avoiding wasting any CPU or memory resources or any time handling them.
Below is an example of a faulty code, where a random value is put before the NodeID

```
3:j5Ijs9kJ3:33:0:1:25:18:-16
```

Since the ESP32 reads the NodeID as “3” instead of “j5IjS0kJ3”, it ignores the string and waits for the next input string.

4.4.4.2 Guru Meditation Error

As mentioned in “4.4.3.2 Guru Meditation Error: Core 1 panic'ed (LoadProhibited). Exception was unhandled.”, the ESP32 struggles when it receives a string that uses a NULL as delimiter after the last value. The temporary solution for this was to include a delimiter at the end of the string. Serial.read() then reads it at “NodeId:0:0:1:\0”. This way, the strtok-function finds a delimiter after the last value, stopping the crash from occurring. This is rather a work-around than a proper solution but was the only solution found to circumvent it without more tools to properly research the problem, in addition to lack of documentation or support online explaining why and how this crash occurs, nor any ways to fix it.

4.4.5 Future extensions and improvements in the Uplink code.

4.4.5.1 Proper Error Handling

A big potential improvement for the code is the implementation of a proper error handling routine, making the ESP32 less prone to crashes or lockups. As of now, if the ESP32 encounters an error it will most likely crash with the “core 1 panic'ed (LoadProhibited).”

error message. This reboots the system, and given that the error persists, it can get stuck in a crashing loop which is unfortunate for the stability of our system. This is a built-in error-handling method called “fail hard” in the ESP32 itself, where all operations halt and the ESP32 reboots to avoid any damage to the system. Considering this, the team must in any future development add some sort of “fail soft” mode, where it in case of a crash-causing error, the microcontroller can circumvent the problematic part, and keep operations as normal as possible, although limited to prevent further errors until the problem is resolved. With less severe errors, the team must implement some kind of “fail safe” methods, where it can skip simple error-prone steps, but still keep operations at a normal level.

Another proposed improvement as a follow up to this idea is to add error report feeding directly to the Firebase database, where in case of an error, the ESP32 uploads a pre-set error message telling which NodeID encountered an error, and possibly what the error is such as missing or garbled values, loss of connection from the gateway or node etc.

4.4.5.2 Short-term optimization plans

Some optimization can also be achieved throughout the uplink code. The team has investigated the possibility of using structs to loop through the variables for each function instead of repeating them for each function, however there were some problems applying it with the code, and as such pushed it beyond the scope of the current system. Using structs will help shorten the code somewhat, making it more readable. This especially makes the string parsing a lot cleaner,

An opportunity the team did not get to explore more is the ability to save the Firebase Host, Auth, Wi-Fi SSID and PASS to the internal memory, which could open up the possibility to prompt the user to input these on a PC via serial communication such as USB, which in turn would allow the user to switch between multiple different networks and databases with a single node without having to change these values via programming software, which is considered a flaw in the uplink.

4.5 Concerning the node and the gateway

The team looked at different types of wireless communications. As an example, BLE (Bluetooth Energy) wireless technology was considered, but fell through at the discovery of

bad signal retention. The Bluetooth Low Energy wireless technology has good data rates at smaller ranges however if there is noise or an absorption material, the signal range falls. The team also looked at the RFM95[56] which is a popular chip for IoT systems, using a sub 1 GHz technology called LoRa. However, TI-MCU's have lower power consumption, as the CC1312R have a wakeup on external interrupt current usage of 150nA, while the RFM95 have a current of 0,2 microamps in sleep mode.

The TI- simplelink sub 1 GHz CC1312R wireless MCU provides a long range and excellent RF performance that results in a robust connection. The MCU is capable of industry low terms of power consumption that enable sensors to operate for many years on batteries. It also answers the security and scalability requirement mentioned, as there are dedicated drivers for cryptography and radio protocols. The CC1352R could be a replacement however this MCU does not have Wi-Fi, and as such there was a decision to go for the cheaper price option without Bluetooth Low Energy.

Regarding network topology, ring topology might be usable, especially more intricate modifications such as the double ring topology. Such systems might be useful, should the communications system of choice be Bluetooth based, or other low range systems. Where every node both can function as a repeater, but also fundamentally parsing on the data through the chain and echoing it back to the node with the query, data requests etc. However, seeing the massive range of the CC1312R and without the need of range expansion through the nodes, a star topology is prerequisite and preferred in the given communications system. The major advantage with the star topology in the given communications chain is bypassing the reliance on every single node in the system, while also reducing the redundant communication through every node as they parse the data through the chain. With the focus of the system being ease of use, reliable, low power and low data transmission system, star topology is by far the best option.

4.5.1 Firmware

The link between the node to the localhost server where the sensor data could be viewed showed no problems during the testing stage. The firmware could have been coded in many

ways, the programmers using mainly experience from their earlier NTNU programming classes. With a duty cycle below 1%, LBT and AFA was not needed in the tested firmware.

4.5.2 Unfinished firmware prototypes

The firmware for multiple sensor nodes and security in the radio communication between the node and the gateway was not completely finished. The different firmware prototypes have their own bugs that break the behaviour of the node and the gateway. Solutions to the respective bugs have not been found this far but a lot of progress have been made on either of the bugs, so a functioning code is a possibility.

4.5.2.1 micreedlightnodefinal and micreedlightgatewayfinal project

The micreedlightnodefinal and micreedlightgatewayfinal prototype projects implements a similar process as wakeonreednodefinal[28] and wakeonreedgatewayfinal[29]. In these prototype projects however, the sensor firmware will wake the CPU either every 2 minutes and send 1 packet containing the reed switch sensor state, a current analog brightness sensor ADC reading and analog ADC sensor reading, or send a packet containing reed, light and ADC sensor data whenever the reed switch sensor changes states. The project contains a bug where the node does not always send the ADC and brightness sensor data. Connecting the node to power starts the code and 1 packet is sent to the gateway containing correct information on the ADC and brightness sensor value along with the reed sensor, battery, and temperature. Sending another packet will however give correct information concerning the reed sensor, battery, and temperature, but not the ADC and the brightness sensor. The ADC and brightness sensor values will be 0 and stay 0 for all packets sent thereafter. Restarting the node will reproduce this behaviour, after configuring the code for the node to only sample the ADC and the brightness sensor. When the reed switch was toggled this behaviour continued. The strange behaviour of the node is as of now unfortunately not yet solved. The conclusion is that the code on the node is responsible for the values set to 0, but the sensor values on the node are correct as they appear in debug mode and in the payload as correct. The problem has been posted to the e2e forum[57], Texas Instruments problem solving forum where the developers can ask questions regarding sub1 GHz and other TI-related questions to TI employees, the 12.05.2020. The current solution is to start over and implement changes one at a time, along with debugging in-between.

4.5.2.2 The Password NodeID pair implementation

The modifications to wakeonreedfinal and wakeonreedgatewayfinal to support NodeID and password pairs can be found in the pdf in the GitHub link[58]. The pdf is a step-by-step guide and will hopefully show what kind of modifications were taken to try to achieve a semblance of safety in the radio communication in the system. The gateway will generate NodeID's and random passwords as pairs and is supposed to at the push of BTN-1(button) send one NodeID-password pair over UART to the connected node. The NodeID and password are dependent on a counter of how many times the button has been pushed. Meaning if the button has been pushed 50 times the next NodeID sent over the UART would be NodeID 51 and password 51. The counter will reset itself if the max number of NodeIDs have been reached. The node would then be placed wherever it would spend its lifetime and the id and password pair would be transmitted back to the gateway by use of the radio whenever a packet is sent and would then be identified. To start the read event of the UART on the node the same BTN-1 must be pushed at the same time as the BTN-1 on the gateway. The modifications in the pdf were done using the wakonreednodefinal[28] and wakeonreednodegatewayfinal[29].
Upsides with such a scheme is:

- This would in a sense give a layer of security where if a node imposter did not have the right password it could not send false information further into the system and notify the client of the system of impostors.
- Gives the client the ability to reset IDs/passwords.
- Turning the node off will reset the password and ID. Implementing tamper proofing hardware that just resets the node could be a good implement in the future such as tamper detection sensors. This is of course counterintuitive to giving the client the ability to reset IDs/passwords.
- Low ram usage - the max number of nodes depends on how good the latency on the gateway is, how much processing the gateway needs for each node. This scheme makes it easy for the gateway to compare 255x2 values and thus it should not be a problem with 255 nodes for each gateway. The max number of nodes will go considerably down if the gateway needs to establish encrypted channels between each node, though a gateway with more ram would fix such a problem.

Down sides:

- Giving the client the ability to reset IDs/passwords.
- Without tamper proofing the node's RAM could be accessed and data falsified.

- Passwords need to be very long as the current 8-bit password is too easy to crack thankfully the length of the payload is the only concern there and therefore a password 32 bytes long is possible with the current Easylink radio configuration of payload length 128 bytes long. Which can be expanded further if using other radio configurations.

The current implementation of this scheme does not work. The data is transferred, and the node does indeed get a NodeID and password although wrong as the NodeID that is supposed to be 1 is NodeID 70. A pdf can be found on the code used to produce such errors at the GitHub link[58].

It must be said that any implementation like password NodeID pair implementation will lower the battery life of the node, such is also true for nodes communicating using encrypted communication with the gateway, and also for any other implementation that makes the CPU go into a longer active mode. Only by fine tuning the firmware behaviour of the node, long battery life can be achieved.

4.6 Security

4.6.1 Pairing

A secure system is only as secure as the weakest link, in this system, it would be the assignment of the ID and password. There were several options to choose from, one would be to ship the nodes preconfigured and the only way to change or update the IDs would be to reprogram the node with the new ID, which is a big hassle and not very user-friendly. Another option to pick from would be to have a separate ‘open’ network that could be used by any node to get a new ID and thus network access. This would require a lot of security checks as well as good network code to handle all these requests. The downside to this is that it would be open to exploits from malicious actors.

The solution that was implemented was a hardware one, the node board would be equipped with its own port for assignment of ID and network access. This port would be required to not destroy any of the boards in the event of user mishandling, like connecting them the wrong orientation. To address that potential issue, the ground was placed on the centre pin and one of the terminal pins, and the Vcc were put on the other terminal pin. If the connector is mirrored the powered circuit will be totally grounded on both sides and the Vcc will be connected to the MCU giving the information that it is mirrored.

Several other issues arose from this solution too, namely both MCUs run on the same voltage level, however since the node is battery powered, the voltage will naturally drop over time. This would spell disaster for the node if it got a 3,3V signal while the battery was at a 2V level, the MCU that was selected or any MCU for that matter, cannot tolerate big differences in pin voltage vs. supply voltage only about 0,3V. To fix this problem an NPN transistor was put between the TX and the RX ports, this would force the transmitted signal from the master node to adapt to the supplied battery voltage from the node.

The result was a circuit that requires a potential hacker or malicious entity to have physical access to the master or the node in question to gain access to the network. This is not considered a problem since the master node will not be accessible to everyone.

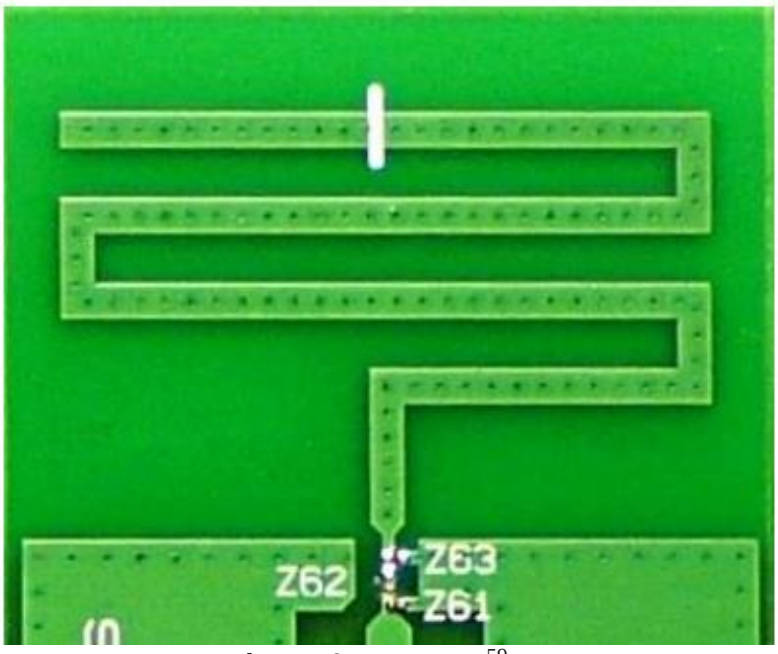
4.7 Hardware

4.7.1 Layout Node

In the final design, there was several key features that were opted for:

- The DN038 PCB helical antenna
- Mounting holes
- Microphone circuit
- Pairing port

First off; the DN038 PCB helical antenna was opted for, this was due to the reference board implementation, the early prototype also sported this antenna and was proved to work. In hindsight another possible solution would be the DN024. This would work better since the required area for that antenna to work is 38 x 25mm, while the available surface area at the top is 21 x 62 mm. it would be inexpensive to increase the height of the PCB to accommodate a larger antenna. The benefits of going for the DN024 antenna is a closer match to 50 ohms without any components, as well as having dual band properties (868MHz and 2440MHz). this makes it possible to use this antenna with Wi-Fi if desirable



The DN024 antenna[59]

The mounting holes provided are 4mm in diameter and made intentionally oversized, the real product would sport smaller holes. This was required since the resolution of a 3D printer is not very high, and this could pose problems with the prototype. The finished product will go for a smaller diameter or a different attachment mechanism for ease of pairing.

The pairing port was designed to make pairing easier and more secure; the team recognized the potential security weakness that may be left if a wireless pairing mechanism were opted for. As it is now the pairing is very cumbersome and would require a disassembly of the PCB should it by some chance be rejected from the network and require a new pairing.

The connector header is still left as it is still a prototype, however a future version should contain terminal ports rather than a pin header. The prototype was not implemented with the terminals because of the lack of space on the circuit board at the point of printing it was still not clear whether a hall effect sensor or a reed switch should be opted for. If a hall effect sensor was opted for, then the sensor would need three wires (ground, Vcc and signal) whereas a reed switch would only require two (Vcc/ground and signal).

The PCB was printed by JCB, this printing job was excellent, and no defects were found on either of the ten supplied PCBs. This is due to following proper procedures for creating circuit boards and following the manufacturers guidelines and the excellent Altium design rule checks. If anything was not according to specifications, they would be caught in those safe nets before ending up on the circuit boards.

4.7.2 Master node

The master node is the central communication device between the nodes and the database, this device has an integrated ESP Wroom[16] and CC1312R[3]. It sports a USB Serial interface, a linear regulator and both TVS and Schottky diode for over voltage and polarity protection. This device is a combined gateway and uplink and therefore will be powered auxiliary by USB; however, this part of the project was not completed due to reprioritization caused by the Covid-19 pandemic and the factory shutdowns that followed. As seen on Figure 46: ESP circuit.

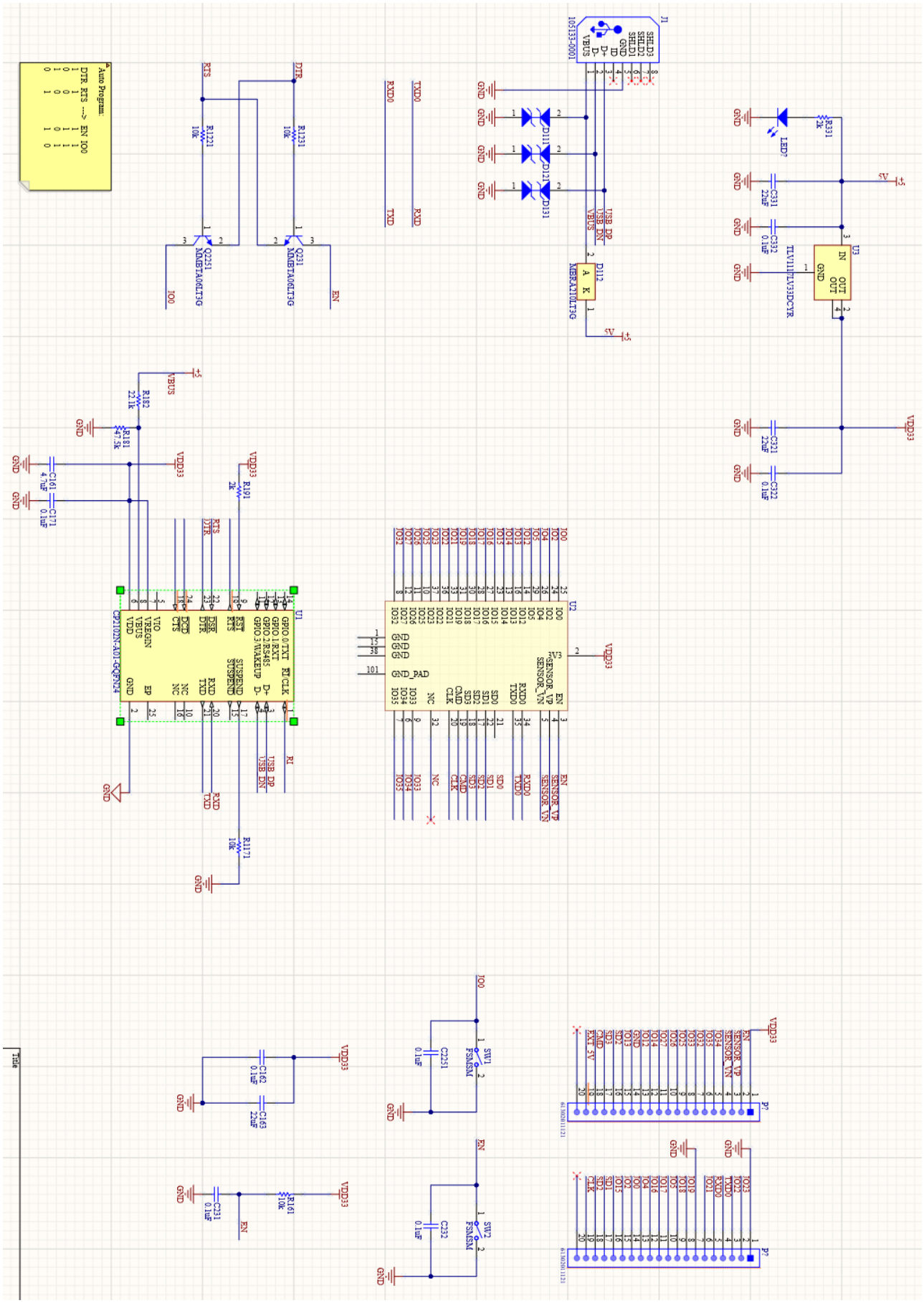


Figure 46: ESP circuit

5 Conclusion

The result is a proof of concept, however not a commercialized solution. The system was successful in monitoring the door state and temperature in a room in Oppdal. With data presented in real time at roughly one second latency, on a web browser in Trondheim. The goal of a low power smart sensor IoT system has thus been achieved, and a proof of concept presented. The current system may transmit the state of a reed switch sensor, along with battery level, brightness of the room, noise and temperature of the CC1312R chip. The only requirement for the end client is a web browser and a Wi-Fi connection.

Several roadblocks were met during the development of the system. Including highly varied programming languages, hundreds of components, and intricate wireless designs. This is not to say that the current conclusion is an unfulfilled bachelor project, however that the design of a complete IoT system is a complex task.

The following recommendations are made for future iterations:

- For simplicity use either a Raspberry Pi as the uplink or an MCU with both Wi-Fi and Sub 1 GHz radio, as the gateway and uplink. If cost efficiency is a factor, it is recommended to keep the modular design with an ESP32 (or similar) as uplink.
- Carefully weight any implementations of security layers and encryption between the node and the gateway. To sustain an encrypted channel between the node and the gateway, power consumption on the node will go up. Also limit the number of nodes per gateway.
- A real time system link may be achieved without any extra cost using Firebase Realtime Database in conjunction with Node.js over a stable Wi-Fi connection. While firebase provides a safe data storage, having additional security layers inside the on-site server (here being Node.js) should be enforced before system rollout.

6 References

[1] '*Long range and Power*, Texas Instruments, accessed 06.05.2020'
http://www.ti.com/wireless-connectivity/simplelink-solutions/sub-1-ghz/overview.html
[2] '*Short Range Devices*, ETSI, accessed 10.05.2020'
https://www.etsi.org/deliver/etsi_en/300200_300299/30022002/03.01.01_30/en_30022002v030101v.pdf
[3] '*CC1312R Datasheet*, Texas Instruments, accessed 20.05.2020'
http://www.ti.com/lit/ds/symlink/cc1312r.pdf?ts=1588763192776
[4] '*Energytrace*, Texas Instruments, accessed 27.05.2020'
http://www.ti.com/tool/ENERGYTRACE
[5] '*MongoDB Atlas*, MongoDB, accessed: 27.05.2020'
https://www.mongodb.com/cloud/atlas/lp/try2
[6] '*MongoDB Licensing*, MongoDB, accessed: 27.05.2020'
https://www.mongodb.com/community/licensing
[7] '*Choose a Database: Cloud Firestore or Realtime Database*, Google, accessed 27.05.2020'
https://firebase.google.com/docs/database/rtdb-vs-firestore
[8] '*RFC 6455 - The WebSocket Protocol*, Tools.ietf.org, https://tools.ietf.org/html/rfc6455, accessed 25.3.2020'
[9] '*Socket.IO*, Socket.IO, accessed 12.4.2020'
https://socket.io/,
[10] '*HTTP Methods GET vs POST*, W3schools.com Accessed 22.2.2020.'
https://www.w3schools.com/tags/ref_httpmethods.asp.
[11] *'express*, npm , accessed 21.3.2020'
https://www.npmjs.com/package/express
[12] '*About Node.js®*, OpenJS Foundation, accessed: 27.05.2020'
https://nodejs.org/en/about/
[13] '*Postman.com*, Postman, https://www.postman.com/, accessed 7.3.2020.'
[14] '*jshint*, npm, https://www.npmjs.com/package/jshint, accessed 15.4.2020'
[15] '*Modules Comparison*, Espressif Systems, accessed 27.05.2020'
https://www.ineltek.com/wp-content/uploads/2018/11/Espressif-Ineltek-Modules-Comparison.pdf
[16] '*ESP32-WROOM-32 Datasheet*, Espressif Systems, accessed 05.2020'
https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf
[17] '*ESP32 Series Datasheet*, Espressif System, accessed 05.2020'
https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
[18] '*Firebase Realtime Database Arduino Library for ESP32*, accessed 05.2020'
https://github.com/mobizt/Firebase-ESP32
[19] R. L. Boylestad, *Introductory circuit analysis.* Boston: Pearson, 2016.
[20] J. Catsoulis, *Designing embedded hardware*. Beijing: OReilly, 2005.
[21] A. S. Sedra and K. C. Smith, *Microelectronic circuits*. New York: Oxford University Press, 2016.
[22] Kompendium: Støy og Forvrengning (elektronikk 2),, 1st ed. Trondheim, Norway: NTNU, 2020.
[23] '*System Design Guidelines for Stellaris® Microcontrollers*, Texas Instruments, accessed 27.05.2020'
http://www.ti.com/lit/an/spma036b/spma036b.pdf?ts=1590621733241
[24] D. Ashby, *Circuit design*. Burlington, MA: Newnes, 2008.
[25] '*HowTo2152*, Jack Olson, Accessed 27.05.2020'
http://frontdoor.biz/PCBportal/HowTo2152.xls
[26] '*AN100 – Crystal Selection Guide*, Texas Instruments, Accessed 27.05.2020'
http://www.ti.com/lit/an/swra372c/swra372c.pdf?&ts=1590056894794
[27] 'Resource explorer, Texas Instruments, accessed 12.05.2020'
http://dev.ti.com/tirex/explore
[28] '*CC1312R node project*, Harald Sæther, accessed 12.05.2020'
https://github.com/Hasjald/CC1312R_LOWPOWERSYSTEM/blob/master/wakeonreednodefinal1.zip
[29] '*CC1312R gateway project'*, Harald Sæther, accessed 12.05.2020'
https://github.com/Hasjald/CC1312R_LOWPOWERSYSTEM/blob/master/wakeonreedgatewayfinal1.zip
[30] '*Product page for ESP32-D0WD'*, Mouser, Accessed 27.05.2020
https://no.mouser.com/ProductDetail/Espressif-Systems/ESP32-D0WD?qs=chTDxNqvsyle7fNmBcU3Gg%3D%3D

[31] '*Product page for ESP32-DevKitC'*, Mouser, Accessed 27.05.2020
https://no.mouser.com/ProductDetail/Espressif-Systems/ESP32-DevKitC?qs=chTDxNqvsyn3pn4VyZwnyQ%3D%3D
[32] '*Structure Your Database | Firebase Realtime Database*, Firebase, https://firebase.google.com/docs/database/web/structure-data, accessed 3.5.2020'
[33] '*HTTP request methods: PUT*, Mozilla , accessed: 27.05.2020'
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT
[34] 'express, npm, accessed 20.05.2020'
https://www.npmjs.com/package/express.
[35] '*Product page Nordic Thingy:52*, Nordic Semiconductor, accessed: 27.05.2020'
https://www.nordicsemi.com/Software-and-tools/Prototyping-platforms/Nordic-Thingy-52
[36] '*Product page for RN2903 LoRa® Mote*, Microchip Technology, Accessed 27.05.2020'
https://www.microchip.com/Developmenttools/ProductDetails/dm164139
[37] '*Miniature Helical PCB Antenna for 868 MHz or 915/920 MHz*, Texas Instruments, accessed 27.05.2020'
http://www.ti.com/lit/an/swra416/swra416.pdf?ts=1591034966601
[38] '*Energytrace results*, Harald Sæther, accessed 13.05.2020'
https://github.com/Hasjald/CC1312R_LOWPOWERSYSTEM/blob/master/energytraceresults.pdf
[39] '*CR2032*, Energizer, accessed 15.05.2020'
https://data.energizer.com/pdfs/cr2032.pdf
[40] '*Smartrf studio*,Texas Instruments, accessed 10.05.2020'
http://www.ti.com/tool/SMARTRFTM-STUDIO
[41] '*Radio Characteristics*, Texas Instruments, accessed 06.05.2020'
http://www.ti.com/lit/an/swra588b/swra588b.pdf?&ts=1589200325899
[42] '*RF-Range estimator*, Texas instruments, accessed 11.05.2020'
http://www.ti.com/tool/RF-RANGE-ESTIMATOR
[43] '*nodeplotlib*, npm, accessed: 01.05.2020'
https://www.npmjs.com/package/nodeplotlib
[44] '*Matplotlib: Python plotting — Matplotlib 3.2.1 documentation*, Matplotlib.org, accessed: 1.05.2020'
https://matplotlib.org/
[45] *'Firebase Predictions*, Firebase, https://firebase.google.com/docs/predictions, accessed 4.4.2020'
[46] 'socket.io, *npm*, accessed 27.05.2020'
https://www.npmjs.com/package/socket.io
[47] "The MIT License | Open Source Initiative, https://www.Opensource.org , accessed 27.05.2020"
https://opensource.org/licenses/MIT
[48] '*LDAP Authorization*, MongoDB, accessed: 27.05.2020'
https://docs.mongodb.com/manual/core/security-ldap-external/
[49] '*Information regarding PCI Security Standards Council*, PCI Security Standards Council, LLC, accessed 27.05.2020'
https://www.pcisecuritystandards.org/
[50] 'Node.Js At Paypal, Medium, https://medium.com/paypal-engineering/node-js-at-paypal-4e2d1d08ce4f accessed 23.2.2020'
[51] '*How NOT to get a $30k bill from Firebase*, Medium, https://medium.com/madhash/how-not-to-get-a-30k-bill-from-firebase-37a6cb3abaca , accessed 14.5.2020'
[52] 'Installation & Setup in JavaScript | Firebase Realtime Database, Firebase, https://firebase.google.com/docs/database/web/start , accessed 15.2.2020'
[53] '*Introduction to the Admin Database API*, Google, accessed: 27.05.2020'
https://firebase.google.com/docs/database/admin/start
[54] '*ESP-IDF Programming Guide'* Espressif Systems, accessed 27.05.2020
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/
[55] '*GitHub, Espressif IoT Development Framework'*, accessed 26.05.2020
https://github.com/espressif/esp-idf
[56] '*RFM95*, HopeRF electronic, accessed 25.05.2020'
https://www.digikey.no/product-detail/en/rf-solutions/RFM95W-915S2/RFM95W-915S2-ND/6564923
[57] '*E2E Forum*, Texas Instruments, accessed 12.05.2020'
https://e2e.ti.com/support/wireless-connectivity/sub-1-ghz/f/156/t/904557
[58] '*CC1312R projectmodifications*, Harald Sæther, accessed 12.05.2020'
https://github.com/Hasjald/CC1312R_LOWPOWERSYSTEM/blob/master/projectmodificationsbachelor.pdf

[59] '*Monopole PCB Antenna with Single or Dual Band Option*, Texas Instruments, accessed 27.05.2020' http://www.ti.com/lit/an/swra227e/swra227e.pdf?ts=1590707218018

7 Appendix

7.1 Project zip files

The project zip files made during the bachelor can be found at the different branches on Github, link https://github.com/Hasjald/Bachelorzipfiles.

7.1 Installing the backend system

The entire chain from database data acquisition until the web host is seen as the backend in this solution. This includes: Firebase Real-time Database and NodeJS. There are various programs and packages that must be installed to correctly run the code supplied in the appendix. These modules, programs and expansions might have received never patches and upgrades in the meantime, which might introduce complications. To truly duplicate the results of this thesis please apply the suggested build version or refer to the developers for advice. All software was run on Windows 10 with a 5th generation Intel processor. Some errors were noticed due to the deviation in clock frequency of the 10th generation of Intel processors. As per 27th of May 2020 it is not recommended to use the Firebase modules for NodeJS until Intel or Google has patched these issues. The processor used with errors was I7-1065G7. With the following error due to the processor of choice:
Assertion failed: new_time >= loop->time, file c:\ws\deps\uv\src\win\core.c

7.1.1 Visual Studio Code

Visual Code Studio April build 2020 [1] was used and functioned as the sole code editor throughout the procedures. Visual Studio Code should per 27th of May be downloaded from Microsoft.

Several extensions were used:

- CodeMetrics (Kiss Tamás)
    - Computes complexity in JavaScript.
- JavaScript (ES6) Code Snippets (Charalampos karypidis)
    - Code snippest for JavaScript ES6 syntax.
- Node.js Modules Intellisense (Zongmin Lei)
    - Autocompletes Node.js modules in import statements.
- npm (egamma)
    - Npm support for VS Code
- npm Intellisense (Christian Kohler)
    - Visual Studio Code plugin that autocompletes npm modules in import statements.
- Search node_modules (Jason Nutter)
    - Quickly search the node_modules folder.
- VS Code for Node.js – Development Pack (Node Source)
    - A starter pack of extensions for VS Code + Node.js.
- PlatformIO
    - Integrates useful IDE functions

7.1.2 Microsoft PowerShell

Microsoft PowerShell v7.0.0 was used integrated with Visual Studio Code as the sole Command-Line Shell. Microsoft PowerShell should per 27th of May 2020 only be downloaded from Microsoft.[2]

[1] Code, V., 2020. *Visual Studio Code - Code Editing. Redefined*. [online] Code.visualstudio.com. Available at: < https://code.visualstudio.com /> [Accessed 26 May 2020].
[2] Installing PowerShell - PowerShell", *Docs.microsoft.com*, 2020. [Online]. Available: https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7. [Accessed: 27- May-2020].

7.1.3 NodeJS

NodeJS 12.17.0 was run and operated inside Visual Studio Code. Node is automatically made available within Visual Studio Code as an extension and should be installed after Visual Studio Code. NodeJS should per 27th May 2020 only be downloaded from OpenJS Foundation. [3]

7.1.4 NPM

Node Package Manager (NPM) is used to download and update NodeJS modules. It is included in the NodeJS installation. NPM should be updated if not the newest version, ensure the version number in the terminal with:
NPM -V

7.1.5 Node Modules

Execution Policy for running scripts in PowerShell on the system (Windows 10) must be enabled before it is possible to use the newly installed modules. One way of ensuring this is changing the policy directly. This may be solved by running PowerShell as administrator with the following command and accept the changes (Y or A for all). [4]
$ Set-ExecutionPolicy RemoteSigned
After performing the operations or during end of development the may be manually reverted to deny script execution again.

After changing the policy, the modules may be installed manually in the terminal through the Node Package Manager (NPM).

$ npm install express
$ npm install socket.io
$ npm install --save firebase

[3] Node.js, *Node.js*, 2020. [Online]. Available: https://nodejs.org/en/. [Accessed: 26- May- 2020].
[4] Installing PowerShell - PowerShell", *Docs.microsoft.com*, 2020. [Online]. Available: https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7. [Accessed: 27- May- 2020].

$ npm install --save firebase-admin

The modules were tested with the following versions: express (4.17.1), socket-io (2.3.0), firebase-admin(8.12.1), firebase (7.14.5).

7.1.6 Google Chrome

Google Chrome[5] version 83.0.4103.61 was used to access the HTML content on http://localhost:4000. The inspect tool was used to monitor errors, bugs and inspect the code served to the end client (CTRL + SHIFT + I). Google Chrome should be used for its excellent W8 JavaScript engine. Other browsers, such as Edge[6], Opera[7] and Firefox[8] was used to ensure broad usability of the IoT solution.

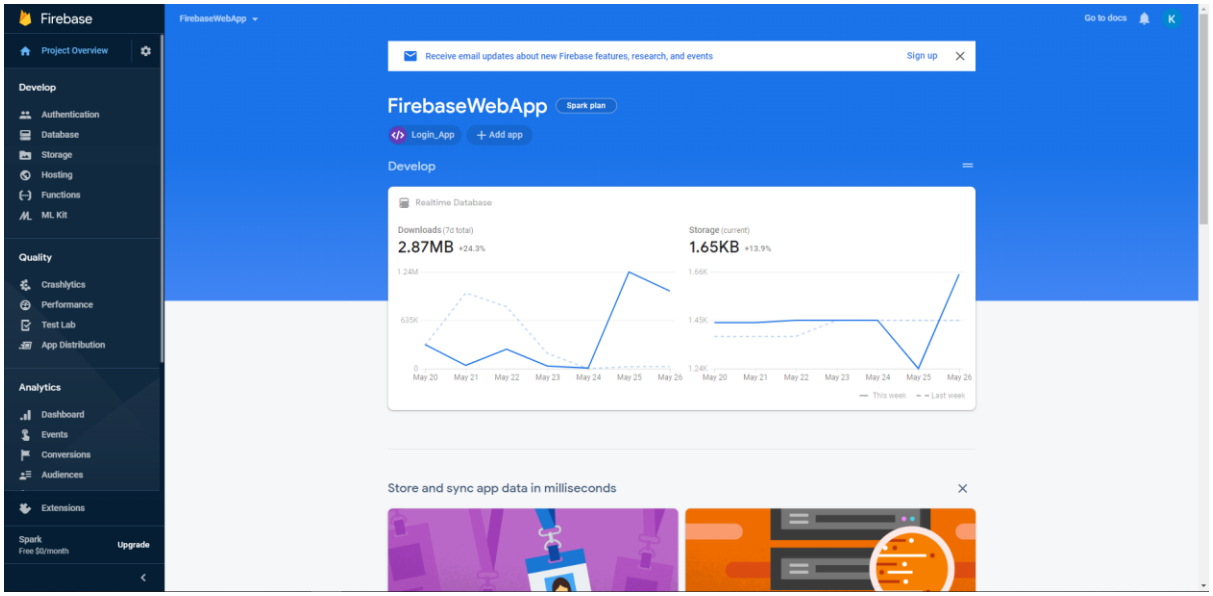
7.1.7 Firebase Real-time Database

Firebase Real-time Database May 21st build was run as SDK and CLI inside NodeJS, as well as the online (always up to date) console inside Google Chrome (Web Browser). The Firebase SDK for NodeJS was used with build 8.12.1. The console may be configured for free by using a Google User.

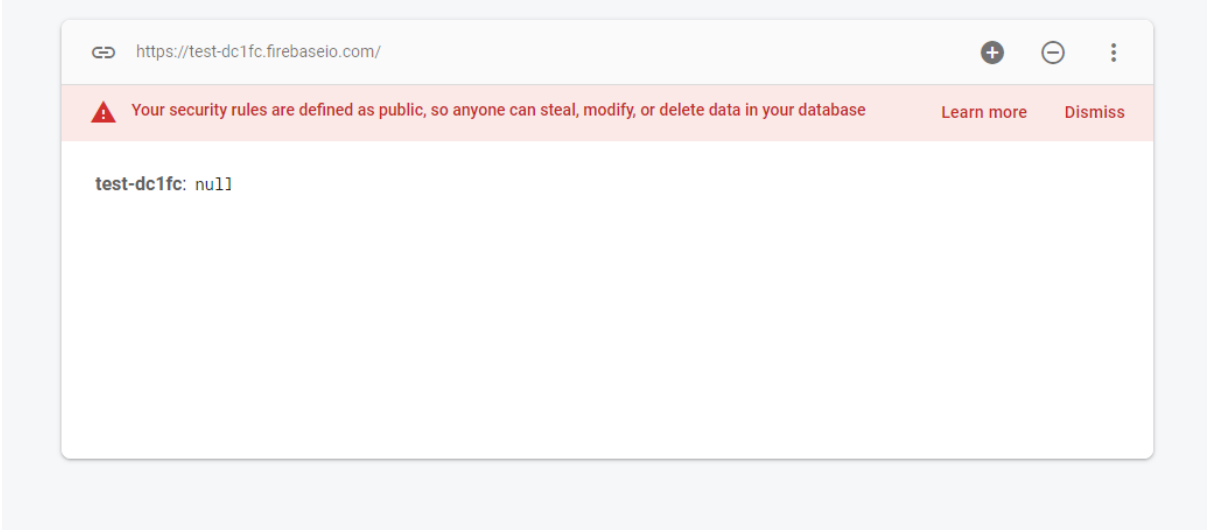
7.1.7.1 Firebase Quick Start Guide

As per 27th May 2020 Firstly the user must create a new project inside console.firebase.google.com inside the new project the user may create a set of databases. It is recommended to keep google analytics enabled for better functionality. The user is then prompted with terms of service, which should be familiarized and read. Lastly the user must select a country of operations. The database(es) should be named wisely as the names will be used during interactions. When successfully configured and installed the user will be greeted by the console:

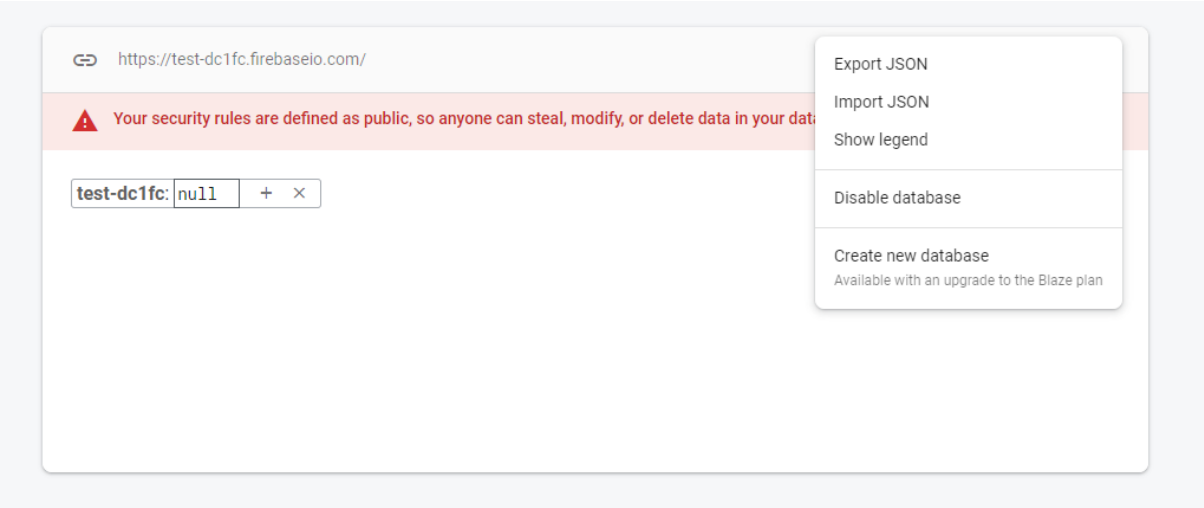
[5] Google Chrome - Download the Fast, Secure Browser from Google", *Google.com*, 2020. [Online]. Available: https://www.google.com/chrome/. [Accessed: 27- May- 2020].
[6] Download New Microsoft Edge Browser | Microsoft", *Microsoft Edge*, 2020. [Online]. Available: https://www.microsoft.com/en-us/edge. [Accessed: 27- May- 2020].
[7] Opera Web Browser | Faster, Safer, Smarter | Opera", *Opera.com*, 2020. [Online]. Available: https://www.opera.com/. [Accessed: 27- May- 2020].
[8] Firefox is more than a browser", *Mozilla*, 2020. [Online]. Available: https://www.mozilla.org/en-US/firefox/products/. [Accessed: 27- May- 2020].



After having the system set up the user must enter the develop tab and open the database section. Develop ▪ Database ▪ Create Real-time Database. The user will be prompted with creating a cloud firestore database, or a real time database. Real-time Database should be selected. Upon creation the database should be started in test mode, these rules may be manually changed at any point regardless. This ruleset opens with full read and write privileges. Using these rules will ease initial connection. After early stages of development, they must be restricted, as further detailed during this thesis. An empty database will then be presented, and the system is successfully set up. New data may be manually added during the setup stages.



The service account is used to connect to firebase in NodeJS, as further detailed during the thesis may be accessed in Project Settings ▪ Service Accounts. The Admin SDK configuration snippet may be accessed, and a new private key may be generated. This key must be safely stored and should under no circumstance be shared or uploaded to online storage unless strictly necessary. When the testing is concluded, and or the database should be taken down for security reasons. It may be disabled manually. Furthermore, offline copies of database data may be downloaded, or manually uploaded in the same menu.



7.1.7.2 Firebase SDK configuration

The Firebase CLI part of the Firebase SDK was manually installed using Node Package Manager.[9] The Firebase CLI may be installed using:

$ npm install -g firebase-tools

Upon installing firebase, a Google user associated with the console was logged in (optionally create a fresh one). The user may authenticate and connect using:

$ firebase login

The firebase was then initiated after a successful install:

$ firebase init

[9] "firebase-tools", *npm*, 2020. [Online]. Available: https://www.npmjs.com/package/firebase-tools. [Accessed: 27- May- 2020].

Visual Studio Code prompted the user with a series of choices, where the following responses were made:

$ > (*) Database: Deploy Firebase Realtime Database Rules

$ > Use an existing project

$ > frbasewebapprlcrowdsolutions20 (FirebaseWebApp)

$ (database.rules.json)

Firebase is now initialized and set up to by default to the selected project for future firebase operations. The project and apps was manually checked to ensure connectivity through:

$ firebase apps:list

$ firebase projects:list

Lastly a manual GET command was issued to test the setup and prove a successful configuration:

$ firebase database:get

7.1.8 Code Composer Studio

Firmware for the CC1312R MCU on the node and the gateway is made with use of the CCS IDE in conjunction with the CC13XX and the CC26XX SDK library[10]. The library is needed since it defines the different pins and power, TI-RTOS as an example needs the SDK library for power dependencies and pin configuration. The installation guide to the newest CCS version can be found online[11]. If one is a beginner to CCS or other Texas instruments tools, Texas instruments resource explorer[12] contains a lot of relevant information and examples for developers to start on. The specific SDK and CCS versions the team used were SDK version 3.40.00.02 and CCS version 9.3.0.

7.1.9 Sensor Controller Studio

Sensor-controller studio is a necessary tool for enabling low power consumption on the node. The CC1312R chip have an internal MCU the sensor controller a 16-bit low power RISC processor, which is a chip able to run standalone to the main core of the CC1312R. Downloading SCS can be done on Texas instruments website[13]. The SCS version the team used was version 2.6.0.

[10] 'CC13xx and CC26xx SDK, Texas Instruments, accessed 27.05.2020'
http://www.ti.com/tool/SIMPLELINK-CC13X2-26X2-SDK
[11] 'CCS installation guide, Texas Instruments, accessed 27.05.2020'
https://software-dl.ti.com/ccs/esd/documents/users_guide/index_installation.html
[12] 'CCS installation guide, Texas Instruments, accessed 27.05.2020'
https://software-dl.ti.com/ccs/esd/documents/users_guide/index_installation.html
[13]'SCS download, TI, accessed 27.05.2020'
http://www.ti.com/tool/SENSOR-CONTROLLER-STUDIO

7.2 Hardware designs

7.2.1 Node schematics

The node schematic is visualized in its entirety from Figure 47 to Figure 66

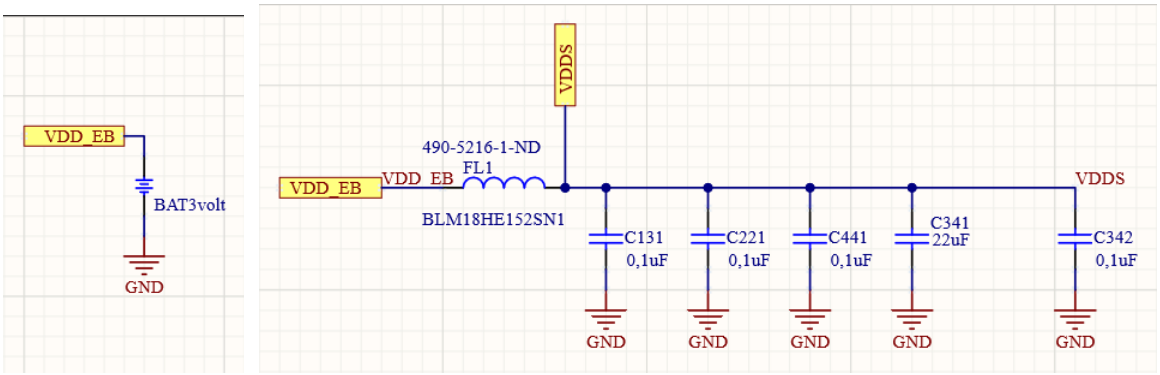


Figure 47: Battery

Figure 48: Node Vcc

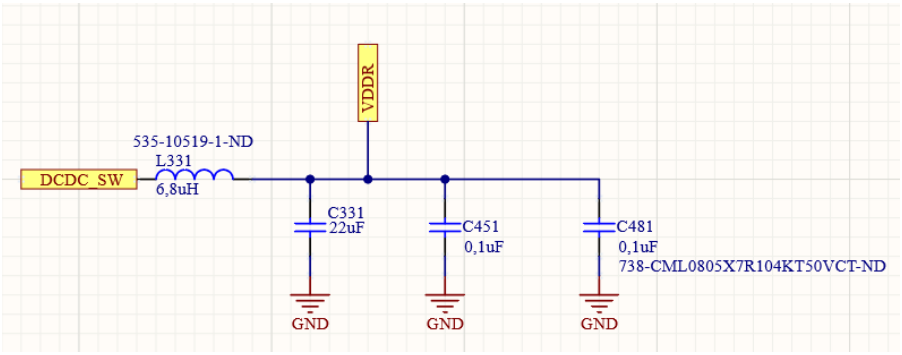


Figure 49: Node RF Power line

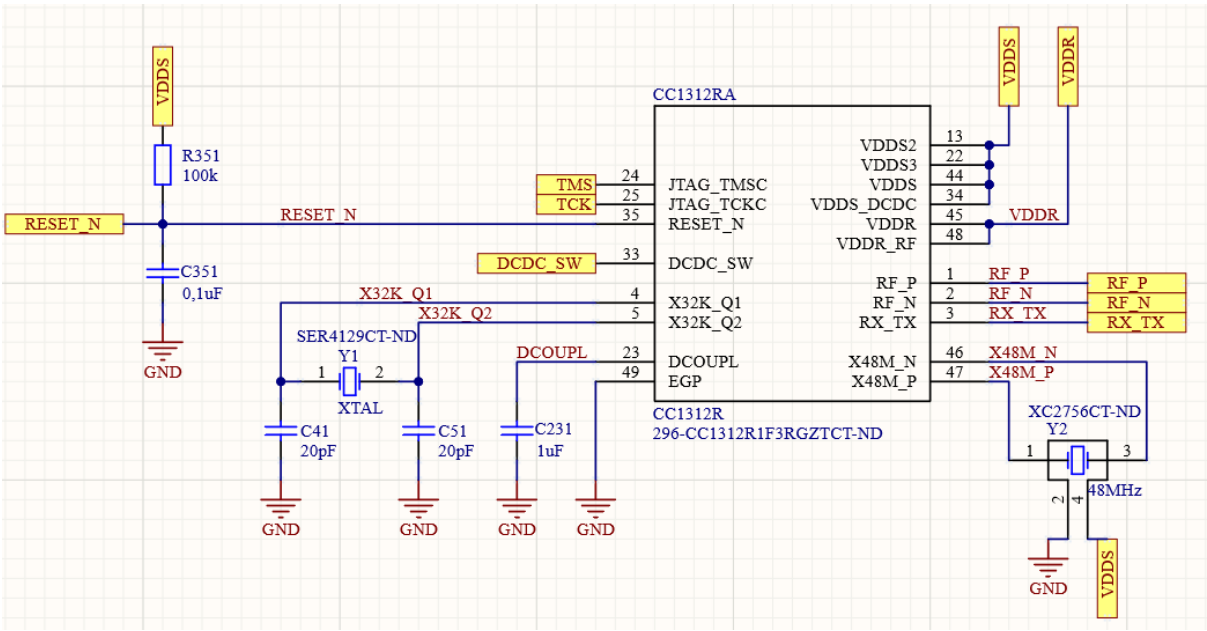


Figure 50: Essential circuitry for Node

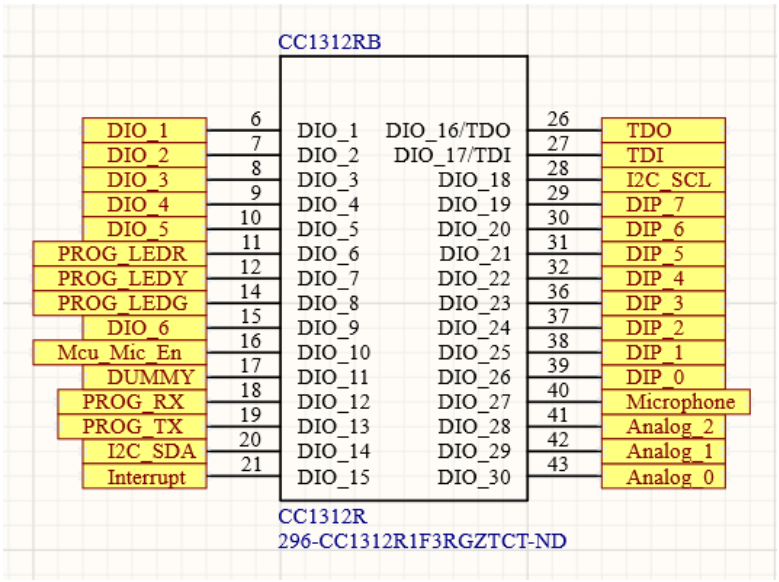


Figure 51: CC1312R Node IO pinout

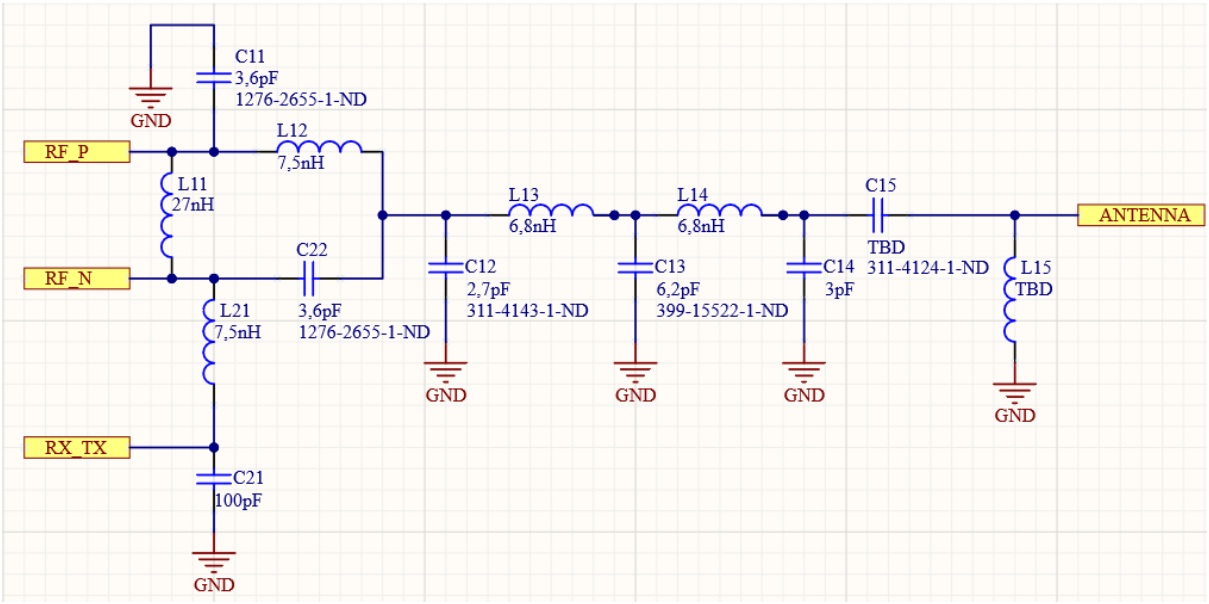
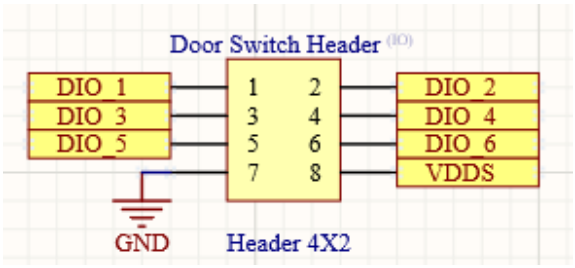
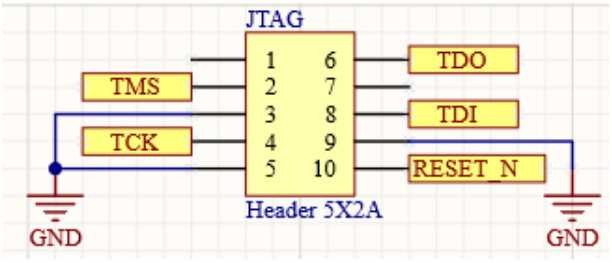


Figure 52: Node RF-bridge and antenna impedance match





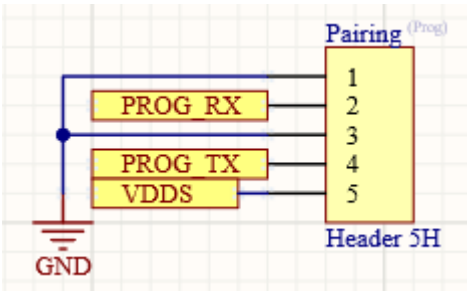
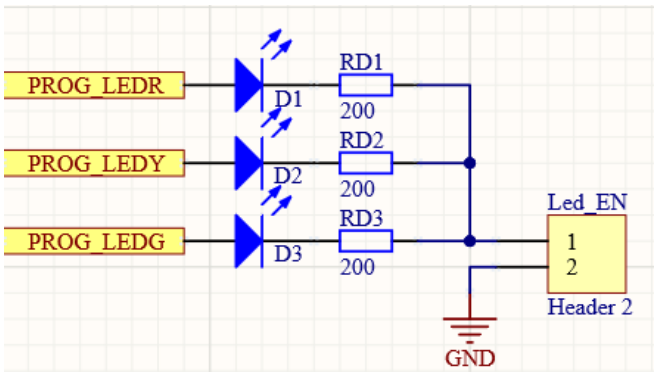


Figure 53: Door switch header

Figure 54: JTAG header

Figure 55: Pairing header



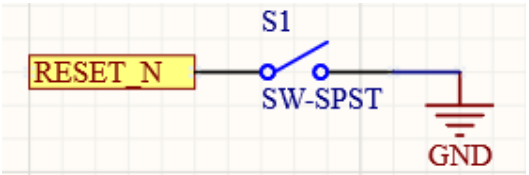
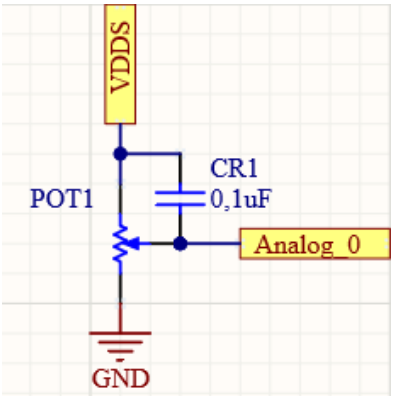
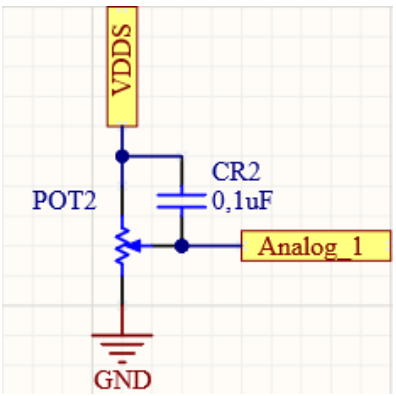


Figure 56: Debug LED

Figure 57: Reset switch





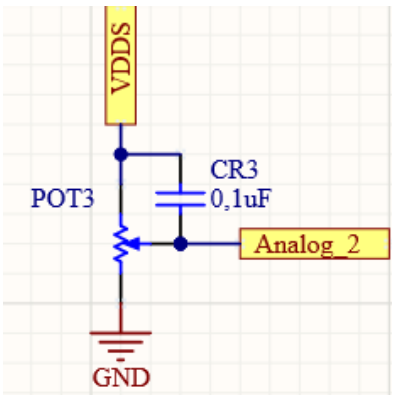


Figure 58: Potentiometer 1

Figure 59: Potentiometer 2

Figure 60: Potentiometer 3

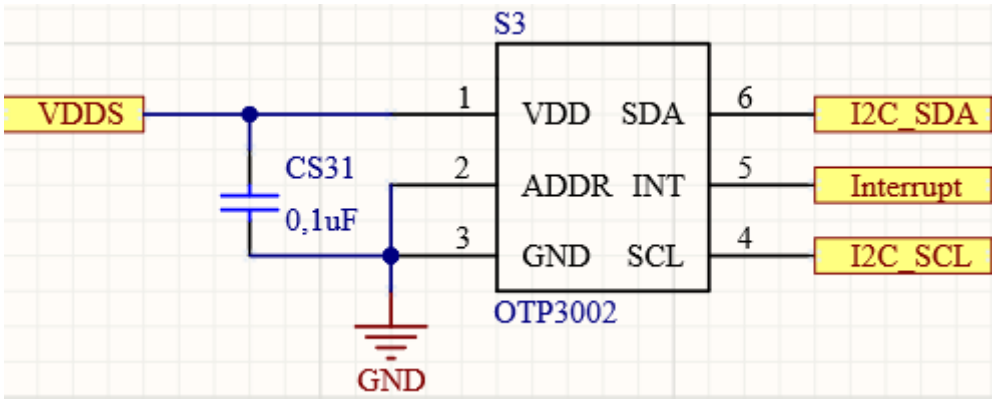
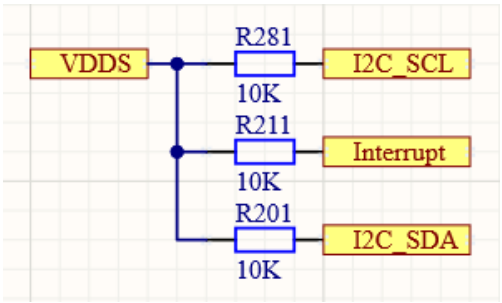


Figure 61: Brightness sensor



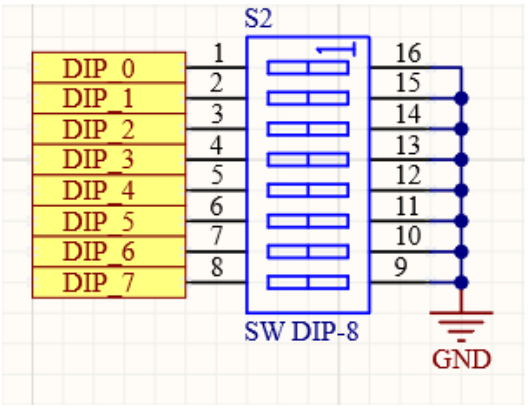


Figure 62: I2C Pullups

Figure 63: Dipswitch module

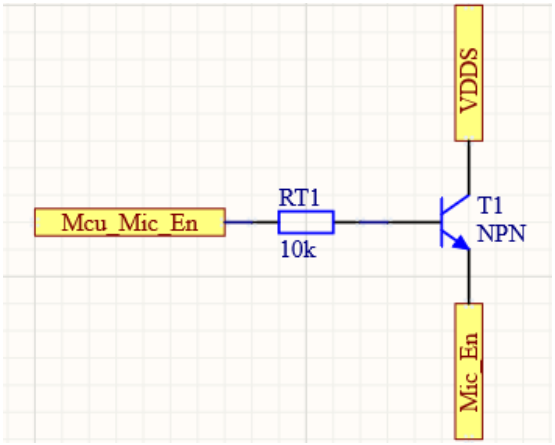


Figure 64: Microphone enable circuit

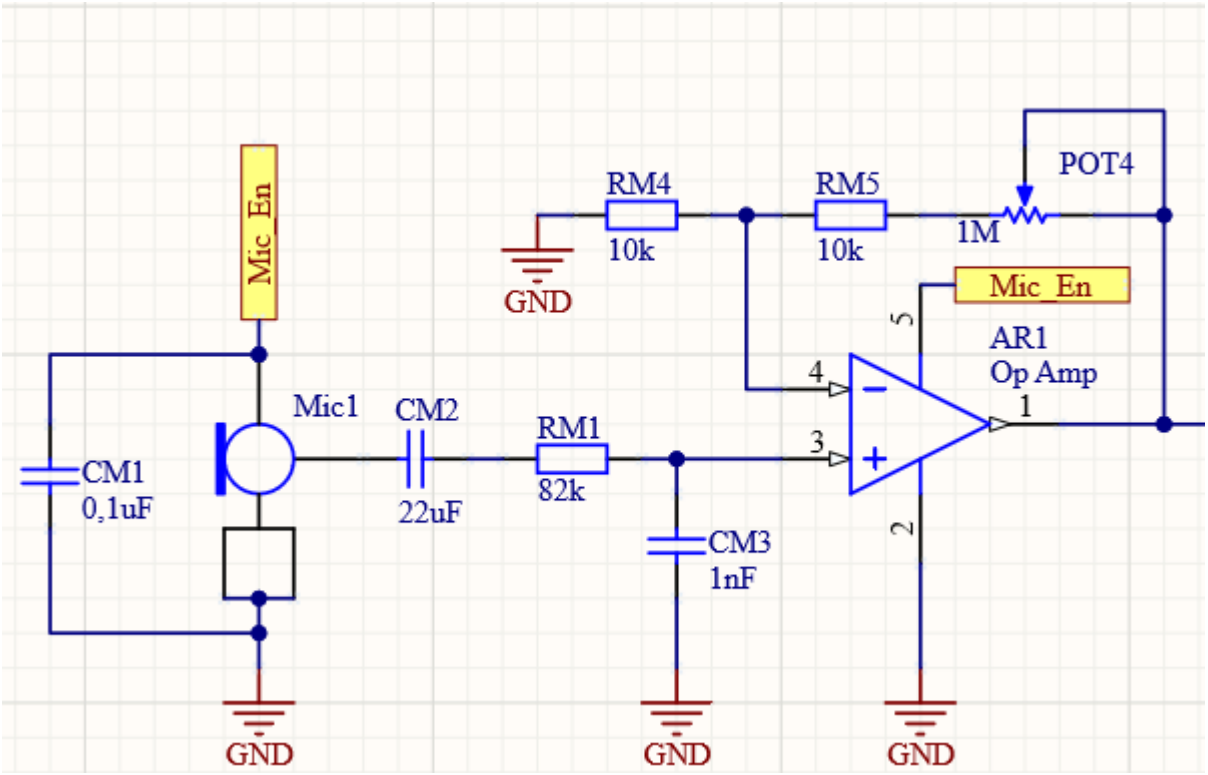


Figure 65: Microphone amplifier circuit

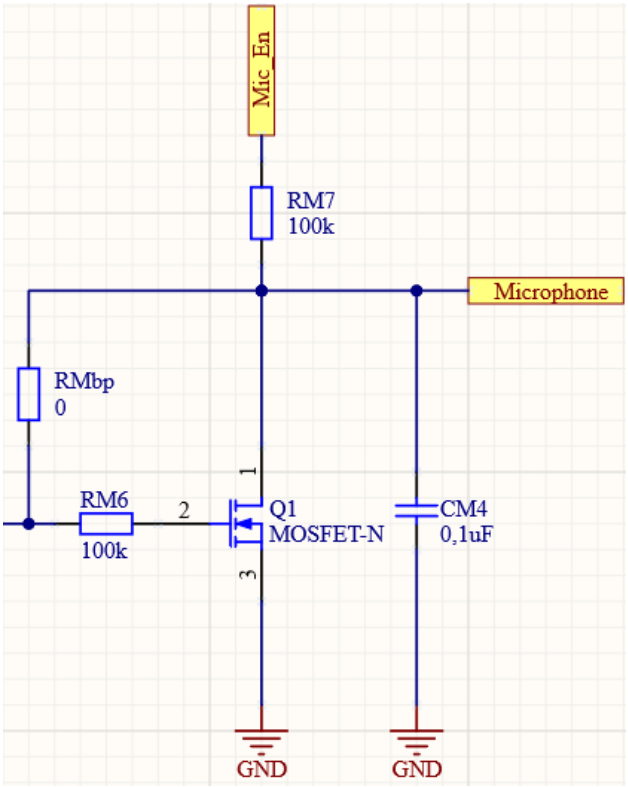


Figure 66: Microphone inverter/bypass circuit

7.2.2 Node Layout

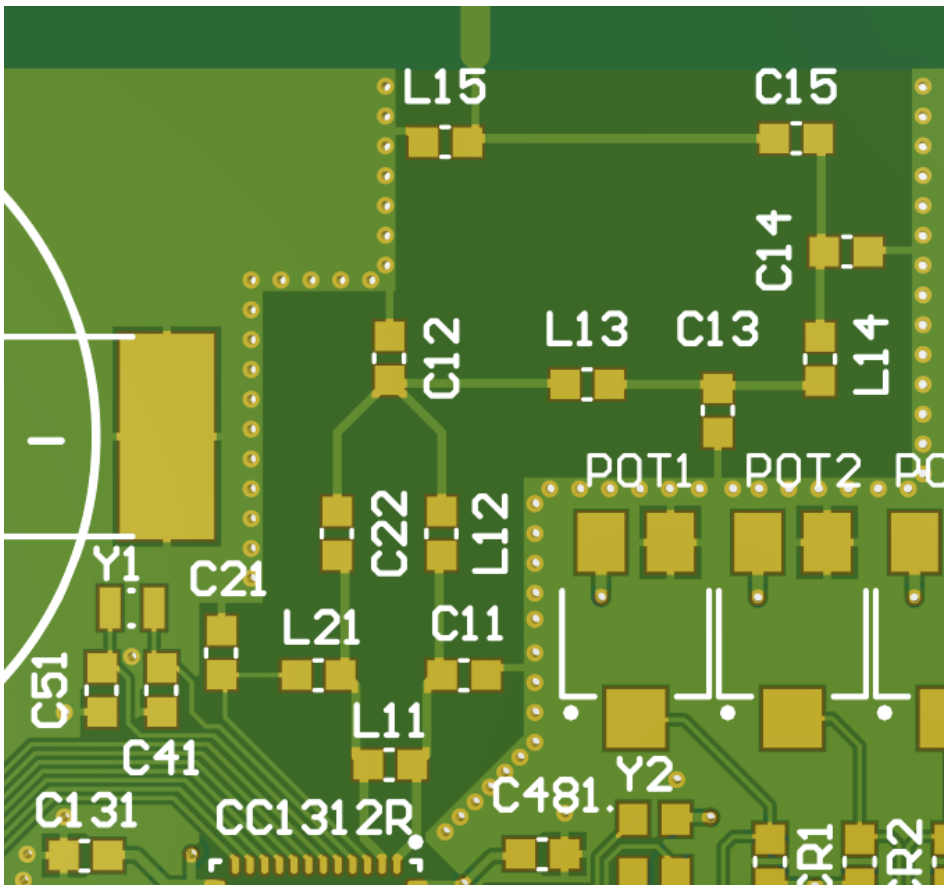


Figure 67: RF-bridge and impedance match

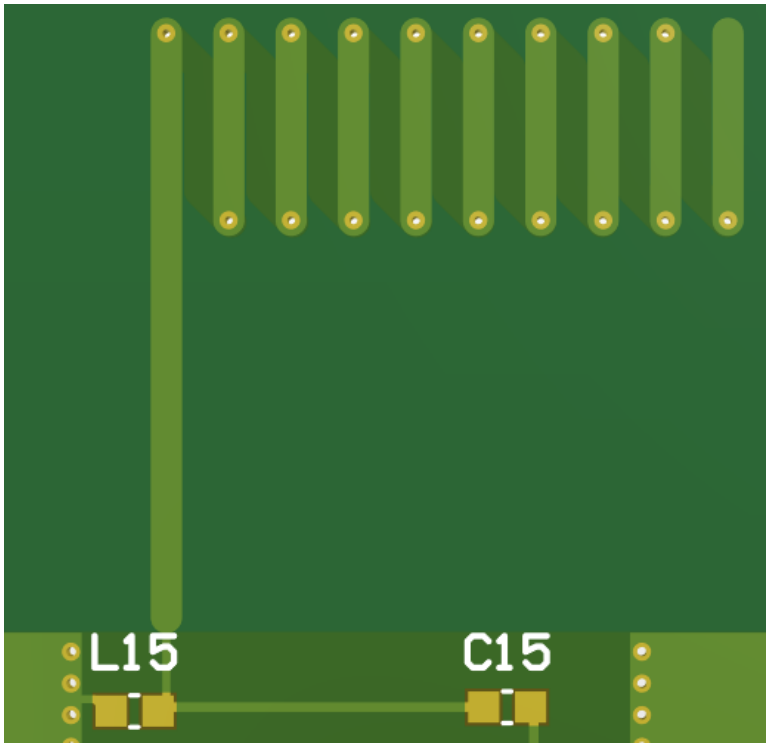


Figure 68: DN038 antenna

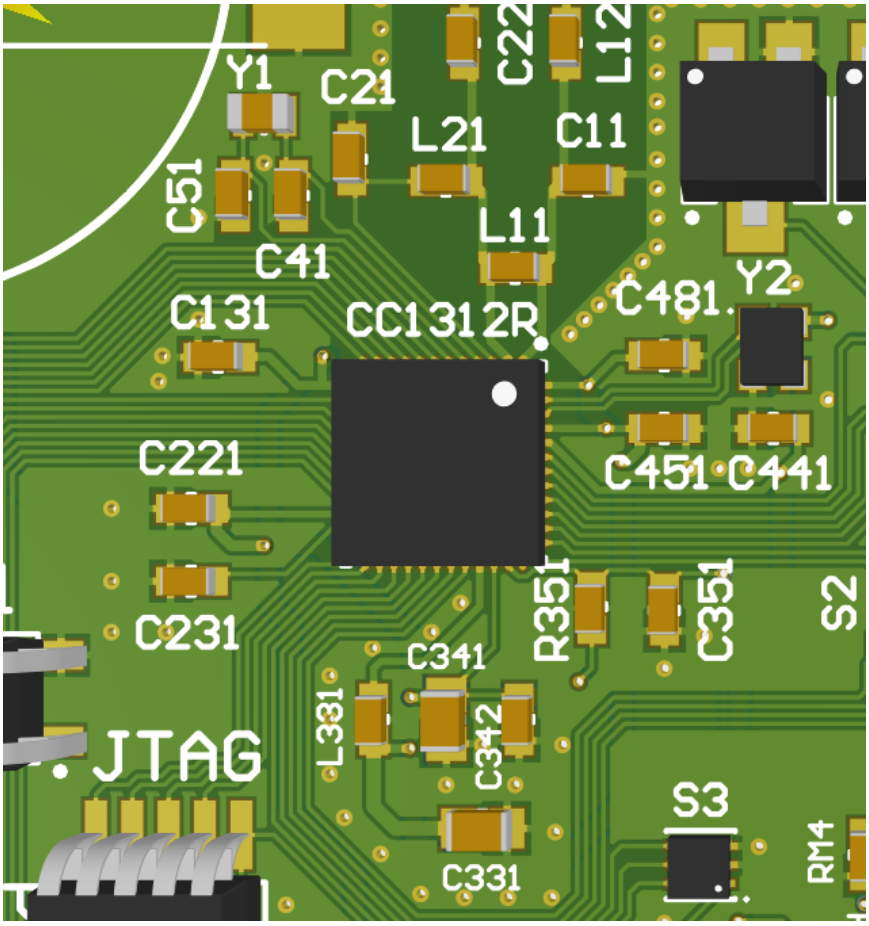


Figure 69: CC1312R Core circuitry

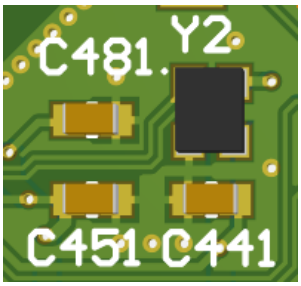


Figure 70: 48MHz crystal oscillator

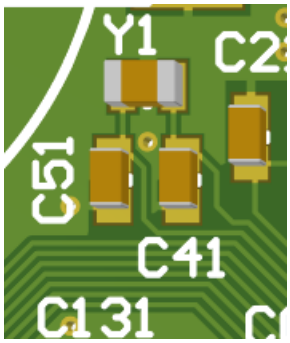


Figure 71: 32,768kHz Crystal

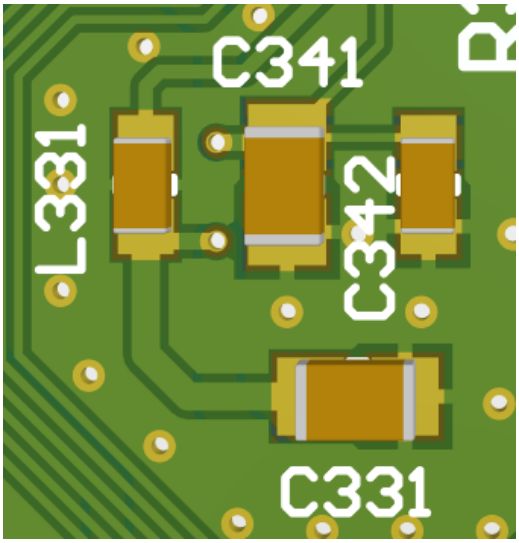


Figure 72: DC/DC Converter circuit for RF

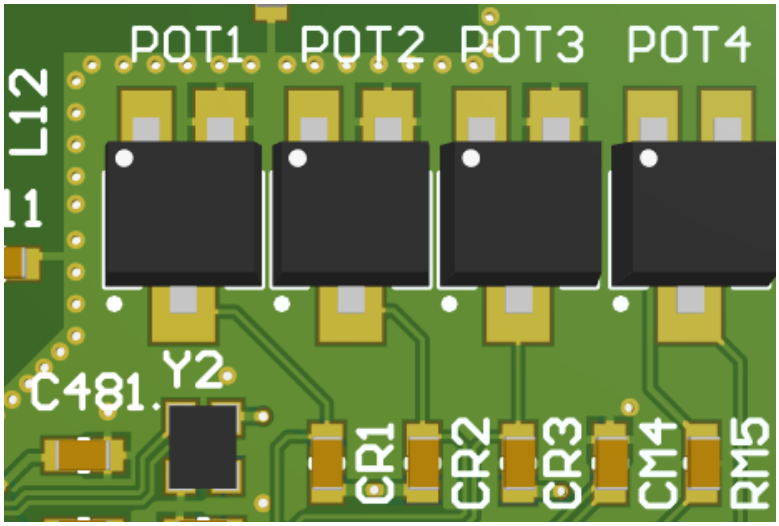


Figure 73: Potentiometers

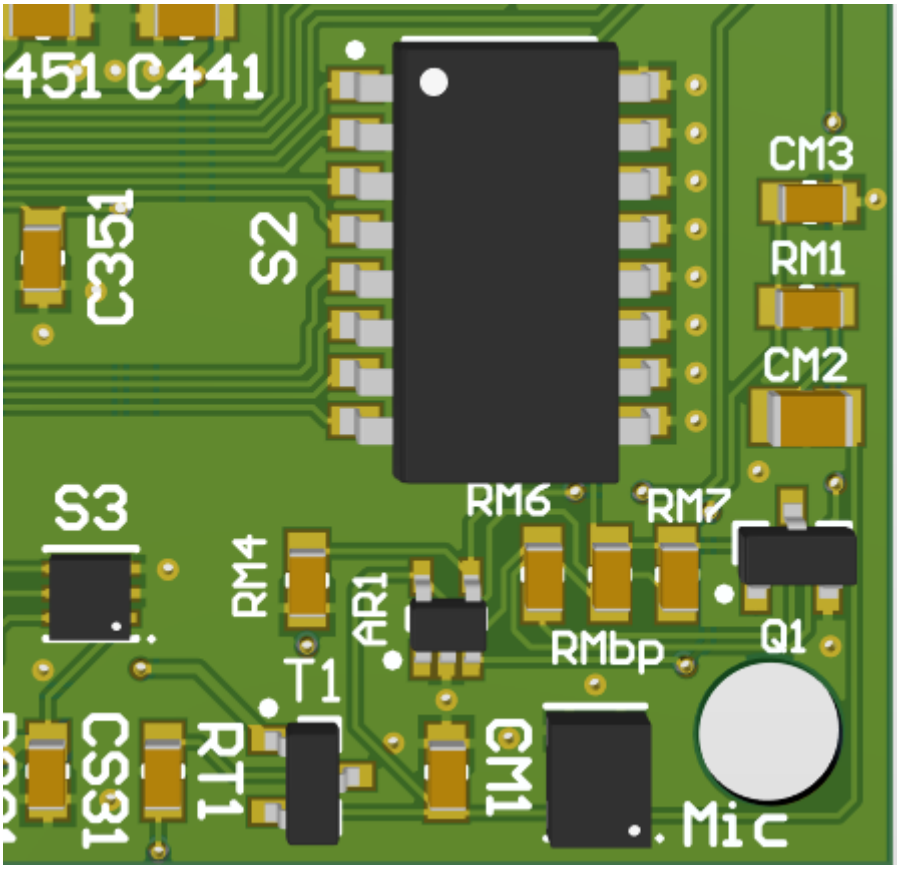


Figure 74: Microphone circuit and DIP switch module

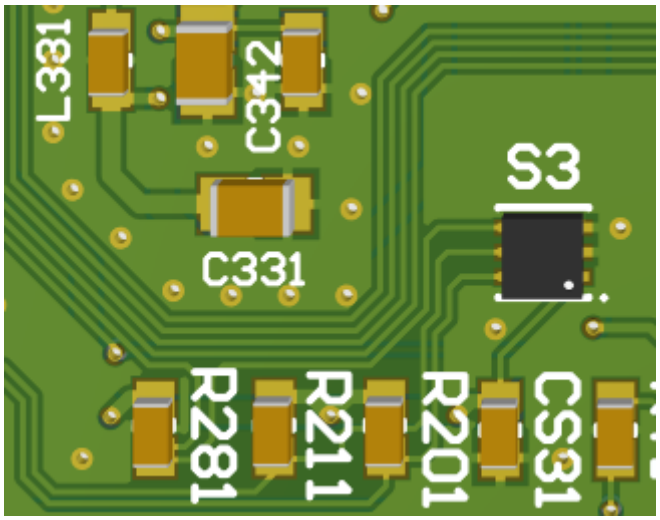


Figure 75: Brightness sensor

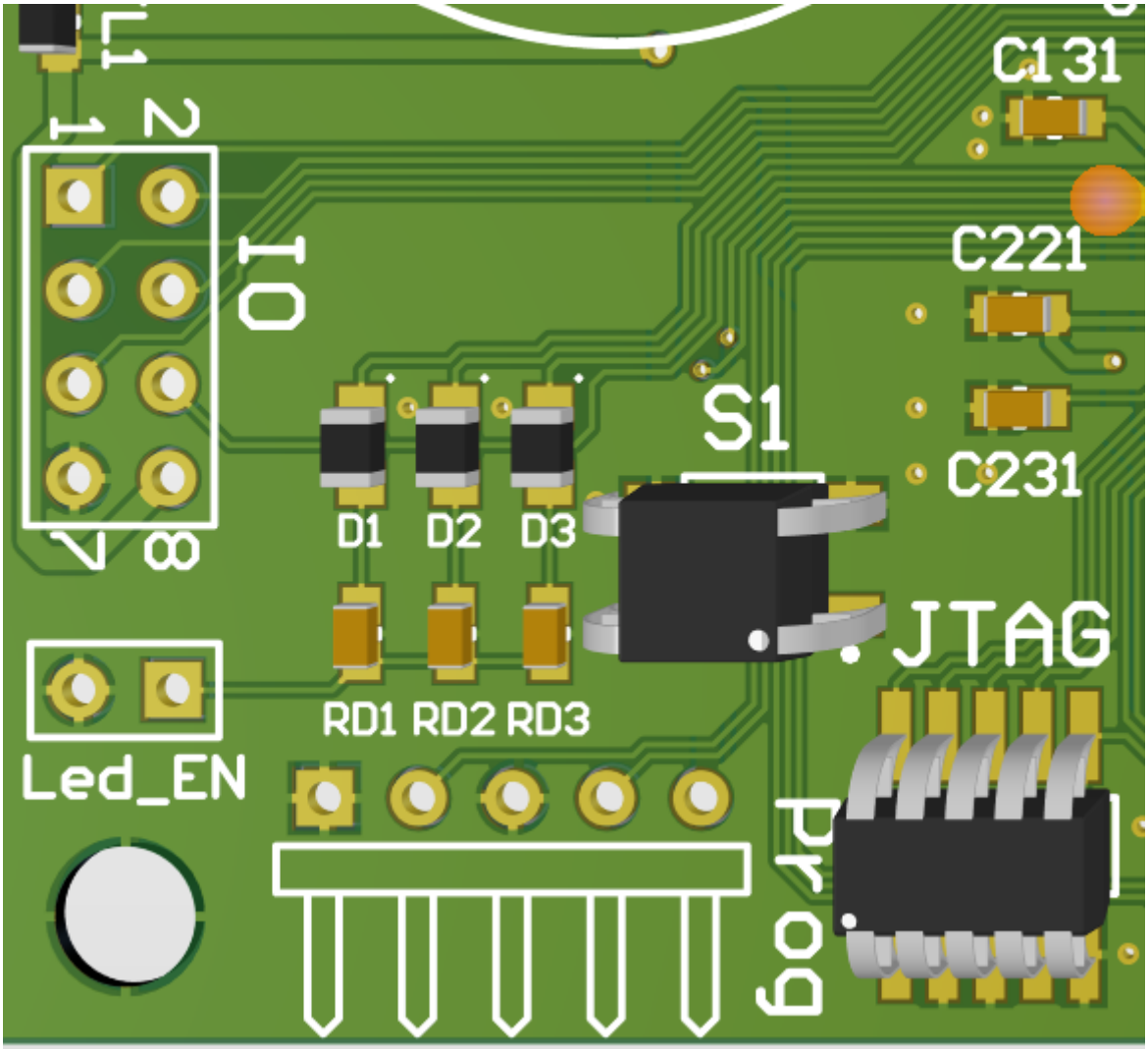


Figure 76: JTAG header, Reset, debug LEDs, LED enable header and door switch header

7.3 Testing the system

Testing the Low power IOT sensor-system

Firmware:

Node - https://bit.ly/36DTdUP

Gateway - https://bit.ly/2zwlRLs

Esp-32 Wroom devkit c - https://bit.ly/3camO9h

Hardware:

- 1 X ESP-32 Wroom devkit-c.
- 2 X LAUNCHXL-CC1312R launchpads.
- 1 X TI ULP-Sensor Boosterpack
- 1X Hyundai 5.5x125mm flat head chrome vanadium screwdriver with magnetic tip.
- 1 X Personal home router with WPA2 encryption and internet access.

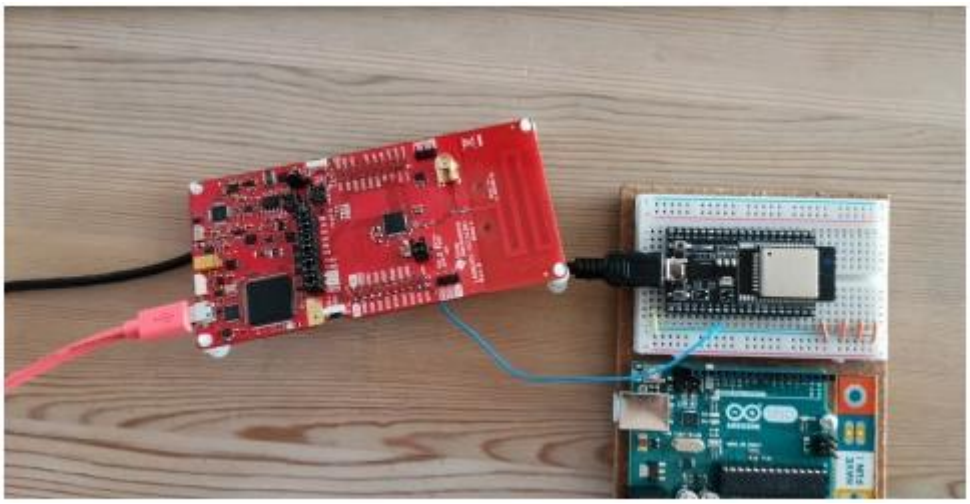


Figure 77: Hardware layout

1. Flash the gateway and node with respective project file, wakeonreedgatewayfinal and wakeonreednodefinal. Flash the ESP-32 and connect pin 16 to DIO1 on the gateway. The node LAUNCHXL should have the ULP-SENSOR Boosterpack attached to the correct DIO-pins, in other words the 5V symbol on the Boosterpack should correspond to the same 5V symbol on the gateway.
2. The two launchpads and the ESP-32 need power through the USB. The ESP32 code needs the personal home router id and network password.
3. Trigger the reeds switch on the node with the magnetic tip screwdriver.
4. Observe as in Figure 78. Firebase will only update if the values coming in are different than what is already there, due to firebase rules.

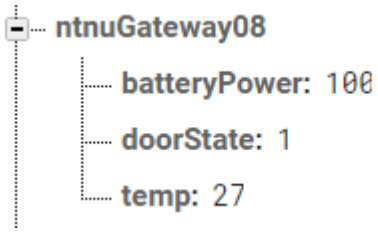


Figure 78: Firebase data

5. The localhost server should display relevant information, see Figure 79:

Figure 79: Localhost data

7.4 Testing node life

Hardware needed:

- 2 X LAUNCHXL-CC1312R
- 1 X ULP-SENSOR BOOSTERPACK
- 1 X Hyundai 5.5x125mm flat head chrome vanadium screwdriver with magnetic tip.

a. Flash the Node project to the launchpad, we then unplug the launchpad and remove jumpers swo,tdi,tdo,tck,tms,rst, txd, 5v and gnd. Do not unplug the xds110 power jumper see Figure 80: CC1312R launchpad

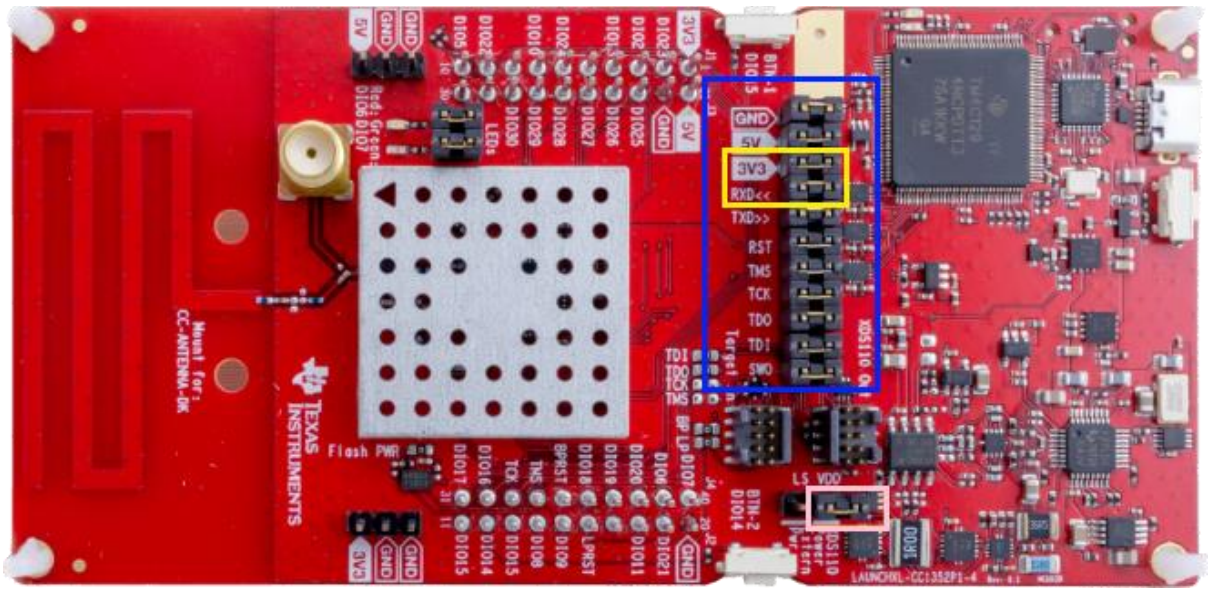


Figure 80: CC1312R launchpad

b. The rxd jumper needs to be connected so the pin does not float and disturb the energy trace readings. Using the ULP-Sensor booster pack in conjunction with the launchpad all sensor power jumpers, except for the ones for the sensors that are to be measured on the boosterpack, needs to be removed Plug in the launchpad and press the energy trace icon by the project builder tool.

c. The energy trace parameters should be default and not need changing; however, going to window and preferences and then ccs->advanced tools->energytrace technology will let one change values on specific parameters, see Figure 81.

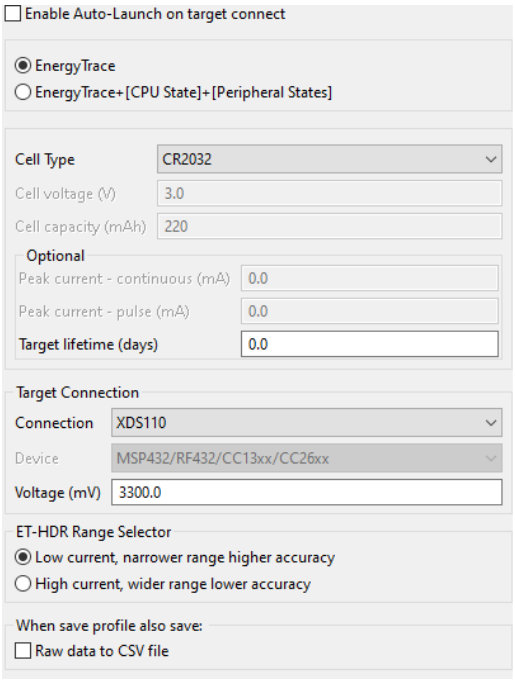


Figure 81: Energytrace configurations

d. Starting energy trace the energy profile panel will appear, configure the timer and press start.

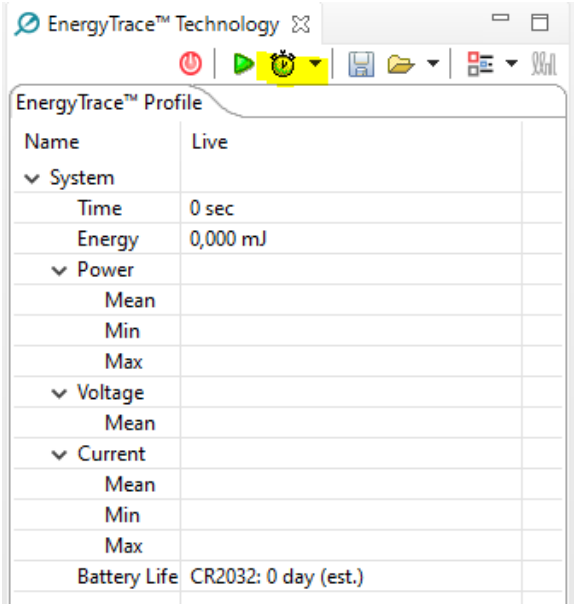


Figure 82: Energytraceprofile panel

e. The energy trace readings will appear in the energy and current tabs.

f. Triggering the reed switch sensor was done with a Hyundai 5.5x125mm flat head chrome vanadium screwdriver with magnetic tip by touching it directly on the reed switch sensor, see Figure 83.

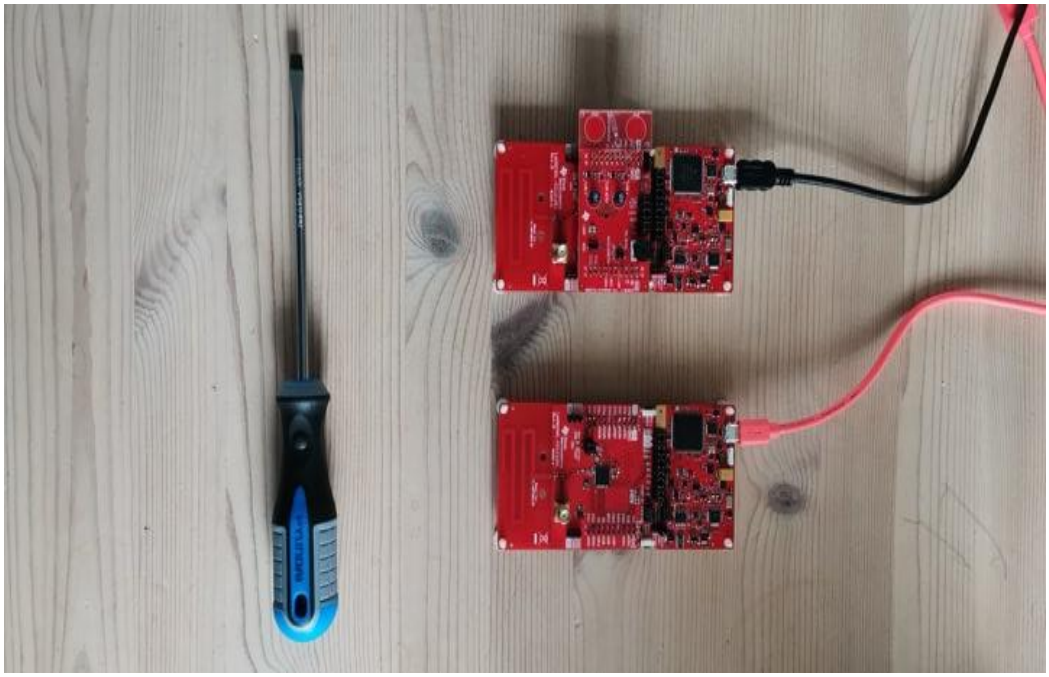


Figure 83: Hardware used

7.5 Setting up and configuring the VSC-project for the Uplink

The installation of VSC is very straight forward. To download VSC, simply visit their homepage, and press the “Download for (Your OS)” button and follow the install wizard. Once VSC is set up, enter the “Extensions” tab on the left side. Enter “PlatformIO” in its search bar, select “PlatformIO IDE”, and press “Install”. The module will now install in VSC and is neatly integrated. You will now probably see a new icon on the left side. If you press this, you will see all the new functions offered by PlatformIO, many of which are similar to the Arduino IDE functions. You will mostly use the “Build”, “Monitor” and “Upload and Monitor” functions when building your projects. Next, to create a new project, enter the PlatformIO home screen, and click on “New Project”. There you can set your project name, select the board you’re working with (in our case, select the Firebeetle-ESP32, it’s a little down on the list), and select your Framework. As the team abstained from utilizing ESP-IDF, it is strongly recommend selecting the Arduino. This also gives a more recognizable framework to work with if you have prior experience using the Arduino IDE or similar software. It is highly recommended to set the location to a more easy-to-find folder when creating your project. The project should now be ready for work, where the “main.cpp” file is your main project. If you want to change some internal settings such as baud rate, CPU-speed reference and more, enter the platformio.ini file and enter the commands there.
It is also highly recommend setting up the shortcut to terminate tasks by opening file -> Preferences -> Keyboard shortcut and searching for “Terminate Task”. This improves the workflow, as by using the shortcut you do not have to click the “stop task” prompt and then press build/upload/etc. again.

To change the firebase host, search the code for #define FIREBASE_HOST and #define FIREBASE_AUTH. These can be found in the initialization part of the code. The FIREBASE_HOST is the input of the firebase address which to send the data to. The FIREBASE_AUTH is the authorization key needed to connect to the database, given by the Firebase webpage. To find these, go to https://console.firebase.google.com/ and enter your project. Click on the cogwheel and select Project settings. There you will find the Authorization key labelled as “web API key”, and the host name as “Project ID”.firebaseio.com

It is also necessary to change the WI-FI that the ESP32 auto-connects to. This is easily done by changing the #define WI-FI_SSID and #define WI-FI_PASSWORD to the corresponding SSID and password of the WI-FI-router.