UNIVERSITY OF AGDER

Development, Deployment & Evaluation of Wireless IoT Devices with Energy Harvesting

Rolf Arne Kjellby
Svein Erik Løtveit
Thor Eirik Johnsrud

SUPERVISORS
Linga Reddy Cenkeramaddi
Geir Jevne

Master's Thesis
University of Agder, 2018
Faculty of Engineering and Science
Department of ICT

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT

Abstract

Automation of indoor climate is becoming increasingly popular for both household and industrial use. Through automation, comfort increases and power consumption decreases. In order to deploy an automation system, sensors are required. This project proposes two wireless sensor nodes based on ATmega328p along with the nRF24l01+ transceiver and nRF52840 with various capabilities in both star and multi-hop network configurations. The designed nodes are fully self-powered through energy harvesting and these nodes are completely self sustainable with no wires, no user intervention during the lifetime of the components. In addition, these nodes do not require any maintenance and can be deployed in remote places. The wireless sensor nodes can be deployed anywhere as long as they are in range of a gateway or nodes that can forward towards a gateway, and as long as there is sufficient light levels for the solar panel, such as indoor lights. Fully functional wireless sensor nodes are designed and tested, and the performance compared of both star and multi-hop topologies. The developed nodes consume less power than what is harvested in both indoor and outdoor environments.

Preface

We appreciate the support and guidance provided by our supervisor Associate Professor Linga Reddy Cenkeramaddi and co-supervisor Geir Jevne. We would also like to thank Wisenet for their support of our project, and Charly Berthod for access to the PV-Lab, as well as assistance with characterization of the solar panels.

The group consists of the following three members:

Thor E. Johnsrud		Svein Erik Løtveit
Thor Eirik Johnsrud	Rolf Arne Kjellby	Svein Erik Løtveit

Grimstad, June 3, 2018

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Explanation
ACK	Acknowledgement
ADC	Analog-to-Digital Converter
API	Application Programming Interface
BLE	Bluetooth Low Energy
BSD	Berkeley Software Distribution
CCA	Clear Channel Assessment
CMSIS	Cortex Microcontroller Software Interface Standard
CNC	Computer Numerical Control
CO	Carbon Monoxide
CO_2	Carbon Dioxide
EH	Energy Harvesting
I^2C	Inter-Integrated Circuit
HFCLK	High Frequency Clock
IAQ	Indoor Air Quality
IC	Integrated Circuit
IDE	Integrated Development Environment
IR	Infrared Radiation
IRQ	Interrupt Request
LED	Light Emitting Diode
LFCLK	Low Frequency Clock
LSB	Least Significant Bit
MCU	Microcontroller Unit
MISO	Master In Slave Out
MSB	Most Significant Bit
MOSI	Master Out Slave In
MPPT	Maximum Power Point Tracking
MQTT	Message Queuing Telemetry Transport
OTA	Over-the-Air

PCB	Printed Circuit Board
PDK	Preview Development Kit
PoC	Proof-of-Concept
PV	Photovoltaic
QoS	Quality of Service
RAK	Rolf Arne Kjellby
RAM	Random Access Memory
RH	Relative Humidity
RSSI	Received Signal Strength Indicator
RTC	Real-Time Clock
SCK	Serial Clock
SCL	Serial Clock Line
SDA	Serial Data Line
SDK	Software Development Kit
SEL	Svein Erik Løtveit
SMD	Surface-Mount Device
SoC	System on Chip
SPI	Serial Peripheral Interface Bus
SS	Slave Select
SSH	Secure Shell
TEJ	Thor Eirik Johnsrud
TWI	Two Wire Interface
UART	Universal Asynchronous Receiver-Transmitter
UiA	University of Agder
USB	Universal Serial Bus
ULP	Ultra-Low Power
UV	Ultraviolet
VOC	Volatile Organic Compounds
WSN	Wireless Sensor Network

Chapter 1

Introduction

1.1 Background

This project is given by Assoc. Prof. Linga Reddy Cenkeramaddi as part of the IKT590 course at University of Agder (UiA) in the Embedded Systems specialization, and will be developed by Rolf Arne Kjellby (RAK), Thor Eirik Johnsrud (TEJ) and Svein Erik Løtveit (SEL).

The motivation for the project is to develop Wireless Sensor Networks (WSN) with Energy Harvesting (EH), and to conduct extensive tests of the various networks in order to present a detailed comparison.

1.2 Problem Statement

Surveillance and automation of homes and industry is becoming increasingly popular in order to obtain a higher level of comfort and potentially reduce energy consumption for heating and ventilation. Several sensor nodes and WSNs exist, however requires recharging or replacement of batteries, or wired power supply.

This report describes sensor nodes with Photovoltaic (PV) EH for indoor and outdoor use, eliminating the need for human intervention throughout the lifetime of the components included in the nodes. The use cases for such a system are many, including smart homes, smart cities and smart agriculture.
Two different sensor nodes are developed and tested in order to determine the optimal solar panel size, transmission interval and communication protocol. Additionally, energy consumption is tested with regard to transmission range, and different network topologies are evaluated.

If time permits, the sensor nodes will be designed for use with supercapacitors, and their performance will be evaluated in order to find the best use-case. Supercapacitors may be required in areas with extreme weather conditions, as common batteries have a drastically lowered voltage and capacity in sub-zero temperatures. They may not, however, be suitable for indoor applications due to a higher leakage current compared to common batteries.

1.3 Project Goals

Main goals and secondary goals associated with this project are outlined in table 1.1.

Table 1.1: Goals

Goal	**Explanation**
MG1	Develop ATmega328p WSN
MG2	Develop nRF52840 WSN
MG3	Energy harvesting
MG4	Research sensors
MG5	Wireless network topologies
MG6	Test and compare the WSNs
SG1	$1.8V$ supply for nRF52840
SG2	Develop the system with supercapacitor, and verify functionality

1.3.1 MG1 - ATmega328p WSN

Develop a WSN based on ATmega328p along with the nRF24l01+ transceiver for wireless communication in the $2.4GHz$ band. The transceiver may be used with Nordic Semiconductor proprietary communication protocol, or with a custom protocol. Printed Circuit Boards (PCB) will be designed and produced, and firmware will be developed.

1.3.2 MG2 - nRF52840 WSN

Develop a WSN based on the nRF52840 Preview Development Kit (PDK). The PDK has an on-board antenna for wireless communication in the $2.4GHz$ band, and can use either Bluetooth 5, Nordic Semiconductor proprietary or a custom communication protocol. If time permits, a PCB will be designed and constructed for the nRF52840 nodes. The Integrated Circtuit (IC) is however recently released and has a long lead time, so the construction of the node may not be achievable during the course of the project.

1.3.3 MG3 - Energy Harvesting

Develop EH circuits for the WSNs. Previous research has been conducted in order to determine the feasibility of various EH sources such as radio frequency and solar, with the conclusion that solar panels are the most viable source of energy for both indoor and outdoor use. Therefore, the EH circuits are designed for use solely with a solar panel. A battery should be implemented, along with a battery charging circuit and a step down voltage converter to obtain stable voltages of the necessary levels. The EH circuit must be able to drive the sensor nodes with no requirement for manual charging or other manual intervention.

1.3.4 MG4 - Sensors

Various sensors should be researched in order to find the most suitable ones for the WSNs. The sensors must be Ultra-Low Power (ULP) and require as little on-time as possible in order to further reduce the energy consumption. The environmental values of interest are:

- Temperature
- Relative Humidity (RH)
- Visible light
- Atmospheric pressure
- Indoor Air Quality (IAQ)

1.3.5 MG5 - Wireless Network Topologies

The WSNs will be configured and tested in both one-hop and multi-hop topologies. Custom firmware will be developed for both topologies, and for both nodes.

1.3.6 MG6 - Testing & Comparison

The WSNs created will undergo extensive testing to determine the indoor and outdoor ranges, as well as energy consumption with various environmental sensors and network topologies. A line of sight range test will be conducted for various configurations of the systems, and for the nRF52840 nodes, both the high bitrate mode and long range mode will be tested. The results will be compared in order to determine the best use-case for each system.

1.3.7 SG1 - 1.8V Operation for nRF52840

The nRF52840 PDK has a supply voltage of $3.3V$, however $1.8V$ should be kept in mind throughout the design process in order to allow for reduced power consumption. This will be developed and tested if time permits.

1.3.8 SG2 - Supercapacitor

By using a supercapacitor instead of a battery, the nodes will achieve a lower cost and may operate better in harsh and cold environments. If time permits, a supercapacitor will be implemented in one or both of the nodes, and its performance will be compared to that of the battery based nodes.

1.4 Report Outline

- Chapter 1 - Introduction
  - Chapter 1 introduces the reader to the problem and describes the background for the project. The goals are stated and elaborated with brief descriptions. The goals describe the main requirements for this project
- Chapter 2 - Theoretical Background
  - Chapter 2 gives a detailed description of the concepts utilized during the project. Theory on photovoltaic harvesting, energy storage, wireless sensor network topologies and communication protocols and technologies are described. Additionally, the software and tools used throughout the project are described briefly.
- Chapter 3 - System Design
  - Chapter 3 presents the solutions researched and developed. This includes a detailed and technical description of the hardware components, design of circuits and PCBs, as well as the firmware for the nodes and gateways.
- Chapter 4 - Testing
  - Chapter 4 gives a detailed overview of the tests conducted during the project, as well as a comparison of the different WSNs and topologies.
- Chapter 5 - Discussion
  - Chapter 5 discusses technical issues and results based on chapter 3 and 4. The discussion is focused on each goal individually, and possible future work and improvements are described.
- Chapter 6 - Conclusion
  - Chapter 6 presents a conclusion of the development process and the performance of the WSNs.

Chapter 2

Overview of Technology and Processes

2.1 Energy Harvesting

The sensor nodes are to rely on energy harvested from ambient sources in order to avoid, or eliminate, the need for manual replacement or recharging of the power cell, effectively reducing the cost of maintenance. Energy harvested from light by the use of PV elements has through previous research proved to be the most viable source.

2.1.1 PV Energy

Converting energy from light to electrical energy is done with a PV cell. The most common type of PV cell consists of two layers of silicon substrate, much like in common semiconductors. Photons absorbed by the silicon may excite electrons from their current atomic orbit through the band-gap of the two materials, resulting in the generation of a charge that can run current through an auxiliary circuit. Only photons with appropriate wavelength will produce current, and the wavelengths that produce current in silicon cells range from approximately $400nm$ to $1100nm$, as can be seen in spectral response chart in figure 2.1.

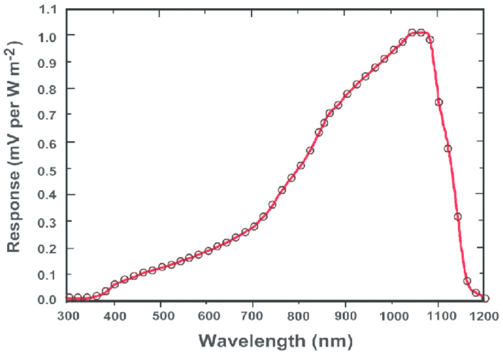


Figure 2.1: Spectral response of silicon cell [1]

The use of PV cells in an indoor environment poses a major challenge, as common artificial indoor lighting has a lower intensity and emit a more narrow spectrum compared to that of the sun. Figure 2.2 shows the wavelengths emitted from common indoor light sources, as well as the sun. By comparing this to the spectral response in figure 2.1, it is clear that artificial light sources emit wavelengths in the lower efficiency region of the silicon cells. PV cells which yield higher efficiency at the lower wavelengths exist, however their cost-to-performance ratio renders them unusable for the purpose of this project. Therefore, low current outputs from the PV modules used for the sensor nodes is a restraint which has to be considered throughout the project.

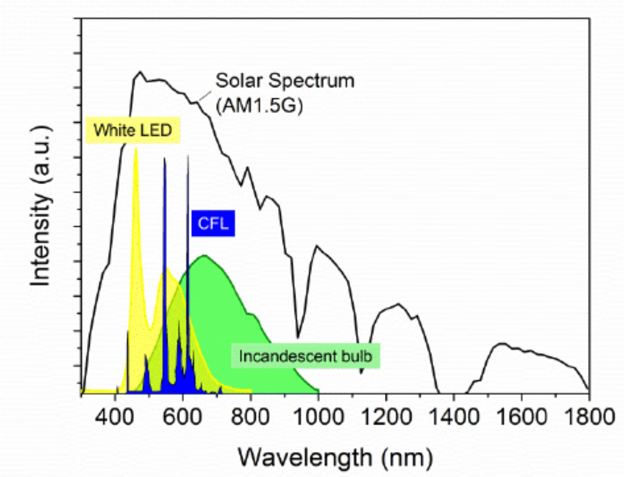


Figure 2.2: Spectrum of different lightsources [2]

PV cells are considered as a high impedance source, and require load balancing to output the highest possible amount of power. This load regulation is referred to as Maximum Power Point Tracking (MPPT), and is usually done in a dynamic manner to ensure that the system performs optimally at all times, as light and temperature is changing through the course of the day. A common method of MPPT is to measure the open circuit voltage of the PV module, and regulate the load such that the voltage is equal to a preset percentage of open circuit voltage. For this method, the maximum power point of the cell has to be known. This is found by exposing a cell to constant lighting, and regulating the load from short circuit to open circuit while measuring voltage and current output. This measurement produce an IV curve, such as can be seen in figure 2.3. As seen in the figure, the power peaks at approximately $4.7V$, which is 78% of the open circuit voltage of $6V$. Most silicon PV cells have their maximum power point at approximately 76% [8].

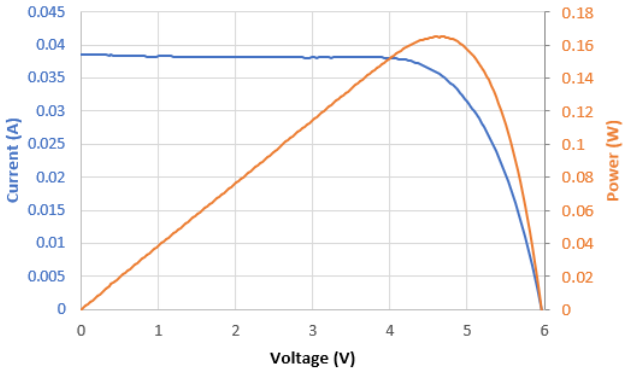


Figure 2.3: IV and power graph of generic Si-panel

2.1.2 Energy Storage

In many indoor environments, the energy available solely from light sources may be inadequate for powering the node directly, such as during nighttime or during the dark periods of the year, however excess amounts may occur during daytime. Energy storage is therefore required, where both supercapacitors and rechargeable batteries are viable options. Supercapacitors can store enough energy for a few transmissions up to a few days of operation, while rechargeable batteries such as lithium ion button cells can store enough energy for weeks to years of operation [9]. Both options do however have their drawbacks. Lithium cells perform poorly in cold conditions and may completely cease to function below $-15°C$. Supercapacitors on the other hand may function very well in cold conditions, but generally have a high self discharge current, often in the range of several μA, which also increase with temperature. This discharge rate of supercapacitors differ between different manufacturers and technologies, but is generally highly dependent on maximum capacitance. The $5F$, $6V$ SCMS32H505MRBB0 supercapacitor from AVX, as an example, has a rated leakage current of $36\mu A$ in 25^oC [10]. The rechargeable lithium ion button cell battery LIR2450 from Multicomp on the other hand, supposedly loses 20% of its rated capacity of $120mAh$ over the course of 3 months at 20^oC, giving a leakage current of approximately $11.1\mu A$ [11].

Energy storage elements usually require conditioning of both input and output power in order to charge in the correct manner, and to supply the circuit with the correct voltage. For low power applications, voltage regulation is usually sufficient, although a combination of constant current and constant voltage based on charge levels is preferable for lithium batteries. Capacitors require overvoltage protection, while lithium batteries require both overvoltage and undervoltage protection to maintain functionality. Output voltage is commonly regulated to a predefined level with a Buck-, Boost-, or Buck/Boost-module to the voltage levels required by the system.

2.2 WSN

2.2.1 WSN Topologies

Two fundamentally different topologies are normally considered in wireless networks, namely multi-hop topology and star topology. A graphical representation of the two topologies is depicted in figure 2.4. Low power wireless communication standards allow for communication between battery powered sensor nodes and gateways with both topologies. The gateway may include both low power radio to communicate with the nodes, as well as handle other communication technologies such as Wi-Fi in order to connect the WSN to the local network or global Internet.

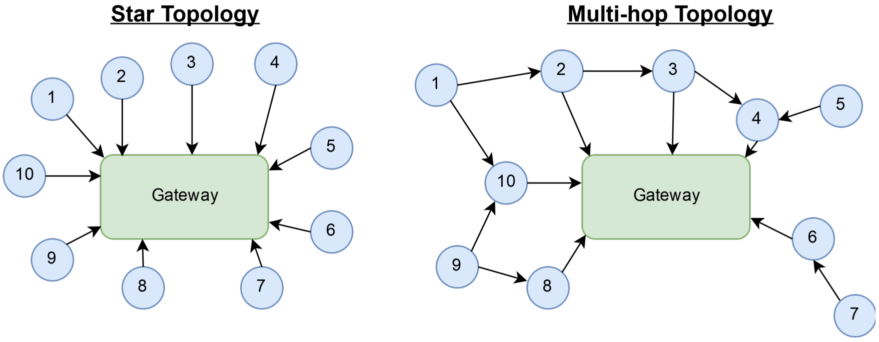


Figure 2.4: Star and multi-hop topology

Star topology is a purely one-hop based topology, where the nodes communicate directly with a gateway [12]. The design and deployment of a star network is straightforward due to its structural simplicity. The one-hop based structure is the main advantage of star topology as no node is required to relay data. No synchronization between nodes is necessary, and higher sleep durations are allowed, resulting in a lower energy consumption. The nodes must, however, be in range of a gateway in order to transmit data, which limits the coverage of the network. Additionally, each cluster of nodes must have at least one gateway which is always active.

On the other hand, in multi-hop topology, nodes are able to relay data from neighboring nodes [12]. In a network, multiple nodes must be on until the data has been sent, resulting in a higher energy consumption due to reduced sleep periods. The relaying of data does, however, permit large areas to be covered as each added node increases the range. Implementing multi-hop topology is difficult compared to star topology. When using timer-based wakeup, all nodes must be synchronized periodically, partly because oscillators generally have poor accuracy and have temperature dependent frequency variations. As a cause of this, a time overlap should be implemented in addition to the periodic synchronization, in order to guarantee that the required nodes are awake at the time of transmission. Consequently, more energy is consumed. The issue called the "energy hole problem" is also a concern regarding network design, which occurs when a single node has to forward packages from multiple neighbors. This results in higher energy consumption in some nodes, and therefore shorter battery life compared to the neighboring nodes, leading to a hole in the transmission line if the battery is depleted. This issue can be avoided in a number of ways, such as increasing the battery capacity of the problematic nodes after identifying them.

Wake-up radios may be introduced to wireless sensor networks in order to avoid unnecessary wake-ups, allowing nodes to dynamically wake up neighbors when data is ready for transmission. More components and possibly more radios are however required, as well as the added complexity to the nodes and the network design [13].

2.2.2 Sensing of Environmental Data

2.2.2.1 Relative Humidity

RH is a measure, represented as a percentage, of the amount of water vapor that exists in the air, relative to amount of air needed for saturation at a given temperature. Warm air can hold more water vapor than cold air, and temperature fluctuations therefore affects the RH. As for indoor climate control, people, buildings and furniture will benefit from having a properly adjusted level of RH. RH regulated from 40% to 60% is considered the most beneficial for human health as it improves general comfort, and may reduce the probability of catching colds, dry skin and irritated eyes [14]. As for buildings and furniture, the correct levels of RH may hinder structural shrinkage, mold and rot.

RH can be measured with a Hygrometer, which can be found in the form of digital ICs. Key figures when selecting such a device is power consumption both during sleep and during measurement, measuring time, as it will contribute to the total on-time of the sensor node, as well as cost and complexity of the IC and its necessary auxiliary circuit.

2.2.2.2 Temperature

Temperature is an important measure when it comes to human comfort in a work environment. Being able to measure temperature accurately enables better control on cooling and heating systems, which may improve health and comfort, while reducing power consumption. What is considered as optimal temperature highly depends on the tasks that are performed, in addition to clothing, RH and other environmental data. Recommended temperature for an office environment may range from $20°C$ to $28°C$ [15]. Environments with more physical

labour may benefit from temperatures as low as 16°C.
Measurement of temperature is straightforward, and temperature is one of the most common parameters to measure with environmental sensing equipment. There are several digital IC's that include temperature measurement.

2.2.2.3 Indoor Air Quality

IAQ is a measure that relies on several parameters. Common influencers of IAQ with regards to human health and comfort include gases such as Carbon Monoxide (CO), Carbon Dioxide (CO_2) and a collection of Volatile Organic Compounds (VOC), in addition to some particulates and microbial contaminants such as animal dandruff, mold and bacteria.

CO is a product of incomplete combustion. High levels of CO deprives the brain of oxygen and can cause nausea, unconsciousness and in worst case, death. Common sources of CO are tobacco smoke, fossil and wood fueled heat sources and car exhaust. Mitigation methods include removing CO sources, air filtering and proper ventilation. CO_2 found in indoor environments is usually a product of human breath. High levels of CO_2 may lead to headaches, drowsiness, mental fatigue and general discomfort [16]. Sufficient ventilation should eliminate issues with CO_2.

VOC gases is a term that covers a wide range of gases emitted from certain liquids and solids, where many of them have adverse short- and long-term health effects ranging from light headaches, skin irritation, damage to nervous system, increased risks of cancer and more. Common sources of VOC gases include paints, solvents, wood preservatives, aerosol sprays, disinfectants, cleaners, air fresheners, glue, hobby supplies and office equipment. The concentration of VOC's in the air is therefore often several times higher inside offices and homes than outside. Mitigation methods include removing sources of VOC where possible, and increasing ventilation to get a sufficiently low concentration of VOC where the sources can not be eliminated. [17]

Measuring gas concentrations is usually quite costly both in hardware and in power-consumption. Most sensors require a heating period to get reliable readings, and some require complex calculations to compensate for humidity, temperature and historical values to get accurate, easily readable values. The selection of IAQ sensors and measurement intervals greatly rely on the amount of energy that is available from the EH system.

2.2.2.4 Atmospheric Pressure

Atmospheric Pressure usually have little or no direct effects on human health or comfort. It can however be used to ensure that a ventilation system is performing optimally, in addition to predicting weather changes and estimate elevation. Atmospheric pressure sensors are included in some environmental sensor IC's.

2.2.2.5 Light Intensity

Light levels in a work environment may have a great effect on performance and general comfort of workers. Too much light consumes unnecessary amounts of power and may be distracting, while too little light may cause drowsiness and headaches, among other things. Sensing light levels can improve automated lighting systems by providing a feedback system. [18]

2.3 Wireless Communication Technologies

2.3.1 Bluetooth 5 Stack

Bluetooth is one of the most widely used communication technologies for short range, low bandwidth applications. It is immensely popular for mobile devices and commonly used for a multitude of multimedia applications. When Bluetooth Low Energy (BLE) was adopted in 2010, Bluetooth also became a viable option for ULP applications being powered by coin cell batteries. While this is most commonly used for smart watches and body sensors, utilization of Bluetooth in WSN applications has become more viable. Short range has however been a limiting factor for the implementation of BLE in battery powered sensor nodes, but that has been greatly improved in the BT5 core specification [7]. BT5 is the next generation Bluetooth technology. It enhances several key features such as range and speed with generally decreased power consumption, although this is device dependent. BT5 also greatly enhances broadcasting abilities and enables mesh-networking. Many of these new features can be used to improve functionality or add new functionality to a WSN. The key differences between BT5 and previous iteration Bluetooth 4.2 (BT4.2) can be seen in table 2.1.

Table 2.1: Key differences in BT4.2 and BT5[7]

Parameter	**BT4.2 LE**	**BT5**
Bitrate	$1Mbps$	$1Mbps$ or $2Mbps$
Max throughput[i]	$0.8Mbps$	$1.4Mbps$
Typical range	$10m$ indoors, $50m$ line of sight	$40m$ indoors, $200m$ line of sight
Long range mode	n/a	$125kbps/500kbps$[ii] up to $1.6km$[iii]
Mesh-enabled	No	Yes

[i] Application throughput without the Bluetooth overhead

[ii] Long Range mode uses 1Mbps with high level coding.

[iii] Distance measured by Texas Instruments with $5dBm$ transmit power [19]

BT5 long range mode utilizes high level coding in order to increase sensitivity of the receiver. Even though bitrate is stated to be $500kbps$ or $125kbps$ in the long range modes, bits are actually transmitted at $1Mbps$ with 2 or 8 symbols per data bit. These high level coding schemes enables the receiver to perform error detection and correction based on received symbols, enabling communication on what is usually considered lossy links without increasing transmitting power. Power consumed while transmitting a set of bits will however still increase, as the transmission will require more time due to the lower bitrate.

The BT5 specification defines a mesh profile [20]. The specified mesh functionality is however based on managed flooding. This means that messages are broadcast from the origin node, and re-broadcast by all the relaying nodes. The flooding of a packet is limited by a "time to live" indicator and a mechanism that hinders re-transmission of previously transmitted packages. Additionally, BT5 mesh also requires some relaying nodes to be active at all times, implying that at least parts of the network will drain large amounts of energy.

2.4 Communication Protocols

2.4.1 Universal Asynchronous Receiver-Transmitter

Universal Asynchronous Receiver-Transmitter (UART) provides full-duplex serial communication between two devices with two signals; Tx and Rx [21]. As UART transmits data in an asynchronous manner, no clock synchronization is required. Instead, the transmitting UART utilizes start and stop bits in transmitted data packets in order to define the beginning and end of each packet. When the receiver reads a start bit, it begins reading incoming bits at a frequency known as baud rate, which is expressed in bits per second (bps). The maximum deviation in baud rate between transmitter and receiver is 10%.

2.4.2 Inter-Integrated Circuit

Inter-Integrated Circuit (I^2C) is a synchronous two-wire interface, using the signals Serial Clock Line (SCL) for clock and Serial Data Line (SDA) for data [22]. Multiple devices can be connected to a single I^2C bus, and as both SCL and SDA are open drain drivers, they require pullup resistors in order to allow high signals. All devices on an I^2C bus is either a master or a slave, where SCL is usually driven by the masters. Both slaves and masters can transfer data on the bus, however the slaves only initiate transfers after being commanded by a master. I^2C has a common clock speed of $100kHz$ in embedded systems.

I^2C has two predefined sequences to indicate the start and stop of a packet. The start sequence occurs when SDA goes *LOW* while SCL is *HIGH*, and the stop sequence when SDA goes *HIGH* while SCL is *HIGH*. During data transmission, SDA may only change when SCL is *LOW*. Each sequence of data is 8 bits, and upon reception of an entire package, the receiver sends an Acknowledgement (ACK) bit. A *LOW* ACK bit indicates that the receiver is ready to receive 8 new bits, while *HIGH* indicates that the transmission is to be terminated, and the master should send a stop sequence.

2.4.3 Serial Peripheral Interface Bus

Serial Peripheral Interface Bus (SPI) is a full-duplex synchronous 4-wire interface, using the signals Master In Slave Out (MOSI), Master Out Slave In (MISO), Serial Clock (SCK) and Slave Select (SS) [23]. The SPI bus has one master and at least one slave, where the master can communicate with all slaves, and slaves can only communicate with the master. SS is used by the master to select which slave to communicate with, and therefore, each slave must have a unique SS signal connection.

2.4.4 Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol which is based on publish and subscribe for communication between devices [24]. MQTT is designed to be efficient, open-source and simple. Due to its lightweight nature and low bandwidth requirements, it is ideal for use in constrained IoT systems. A WSN gateway can publish sensor values to a topic, for example "uia/room10/sensor_value". The topic is not preconfigured in the MQTT server, allowing any host to create a new topic at any time by simply publishing data to it. A host, in this case the IoT server, can subscribe to said topic in order to gather the sensor data.

MQTT has three levels of Quality of Service (QoS), all of which are described in short in table 2.2 [25].

Table 2.2: MQTT QoS types

QoS Type	**Function**
QoS 0	At most once: No ACK from receiver, often referred to as "fire and forget"
QoS 1	At least once: Sender stores the message until ACK is received. If no ACK is received, the message is published again. The publish message can be transmitted multiple times.
QoS 2	Exactly once: Guarantees that each message is received once by the receiver. This is the safest and slowest QoS type.

The publish and subscribe messaging pattern of MQTT requires a message broker. Mosquitto is an MQTT broker commonly used in Linux systems, and can be installed and run on low-cost embedded systems such as Raspberry Pi [26].

2.5 Arduino Environment

Arduino is a simple open-source platform for both hardware and software. Its easy-to-use structure provides a highly convenient tool for Proof-of-Concept (PoC) design and prototyping. Arduino provides an Integrated Development Environment (IDE), which is a cross-platform application written in Java. It supports both C and C++, with the addition of some Arduino specific syntax [27]. An Arduino code must contain a minimum of two functions, namely the *setup()* and *loop()* functions. *setup()* is called only once, when the Microcontroller Unit (MCU) is turned on or when reset. It is commonly used for initialization of variables and other settings. *loop()* is run repeatedly until the MCU is powered off, or enters deep sleep.

In order to utilize the functionality of Arduino, the MCU must be flashed with a bootloader. The Arduino bootloader is a program which permits the uploading of code without additional hardware, simplifying the flashing process. Additionally, the IDE incorporates "avrdude", which converts executable code to hexadecimal encoding, which is then flashed onto the MCU [27].

The Arduino environment offers pre-written libraries for most of the commonly used sensors, wireless radios and other external devices. The libraries are written either by manufacturers, retailers or community members, and can be used free of charge.

2.6 Software and Tools

The various software and tools used throughout the development and testing during the project are outlined below.

2.6.1 Software

- Altium Designer 17.1
    - Altium Designer is a powerful and efficient schematic and PCB layout tool developed by Altium Limited [28].
- Segger Embedded Studio V3.34
    - Segger Embedded Studio is an all-in-one solution for managing, building, testing and deploying embedded applications [29]. It offers a simple project generator, a project manager, source code editor, C/C++ compiler, an integrated debugger and direct J-Link integration. Separate versions exist for ARM and RISC-V MCUs. Segger Embedded Studio requires no license for development with microcontrollers from Nordic Semiconductors
- Arduino Software 1.8.5
    - The Arduino Software is an open-source IDE written in Java. It is used to write and upload code to the hardware.
- Autodesk Inventor 2017
    - Inventor 2017 from Autodesk is a program used for 3D modelling and visualization, simulation and documentation of 3D models [30]. Autodesk provides a student licence that makes it free to use for academic purposes.
- Autodesk Fusion 360
    - Fusion 360 from Autodesk is a all-in-one environment for 3D modelling, simulation and documentation, in addition to supply tools for milling of parts, e.g. PCB and solder stencils [31]. As with Inventor, Fusion 360 is free to use for academic purposes.
- Tera Term 4.97

  - Tera Term is a free software terminal emulator with support for serial port communication and TCP/IP protocols such as telnet and Secure Shell (SSH) [32]. The software is open source under the Berkeley Software Distribution (BSD) License.

- Eclipse Mosquitto 1.5
  - Eclipse Mosquitto is a lightweight MQTT broker suitable for most hardware, ranging from low power embedded devices to powerful servers.

- Hass.io
  - Hass.io is an open source home automation environment powered by Home Assistant, and is suitable for use on a Raspberry Pi [33].

2.6.2 Tools and Instruments

- nRF5 SDK v14.2.0
  - Nordic Semiconductor provides a software development kit which provides drivers, libraries, example codes and more for the nRF52 and nRF51 Series of microcontrollers.

- Segger J-Link EDU
  - The J-Link is a debugger used to flash compiled application code into the MCU memory. It interfaces to the MCU via the JTAG serial protocol.

- Fluke 115 Multimeter
  - The Fluke 115 Multimeter is a hand-held instrument that can measure AC and DC voltage, AC and DC current, resistance and indicate short-circuits [34].

- Keithley 2110 5 1/2 digit Multimeter
  - The Keithley 2110 Multimeter is a benchtop instrument that can measure AC and DC voltage, AC and DC current, resistance, frequency, capacitance, temperature and diodes, and can also log data in .CSV format via a Universal Serial Bus (USB) interface [35].

- Keysight InfiniiVision DSOX2002A Digital Storage Oscilloscope
  - The Keysight InfiniiVision DSOX2002A is a benchtop digital 2 channel oscilloscope with frequency range up to $70MHz$ [36].

- Keysight B2985A Electrometer/High Resistance Meter

  - Keysight B2985A is a benchtop electrometer which measures small currents with $0.01fA$ $(0.01 \cdot 10^{-15}A)$ resolution [37].

- CEL Robox Dual
  - CEL Robox Dual is an easy-to-use 3D printer utilizing the cartesian coordinate system with X, Y and Z axes [38]. The printer uses dual filament extruders and nozzles, allowing for the use of two different materials or filament colors with the DualMaterial^{TM} head, or quicker infill with the QuickFill^{TM} head. It has a heated printer bed with good adhesive features, and proprietary software for slicing 3D models and controlling the printer.

- Snapmaker
  - Snapmaker is a 3-in-1 prototyping machine with the ability to print 3D models, engrave with laser and carve or engrave with Computer Numerical Control (CNC) [39]. It utilizes the cartesian coordinate system, and has a heated bed for 3D printing. The laser module uses a $0.2W$ blue laser. Snapmaker provides proprietary software for both 3D printing and CNC/laser engraving, namely Snapmaker3D and Snapmakerjs.

- Mooshimeter
  - Mooshimeter is a multimeter developed by Mooshim, which uses BLE technology for connecting with smartphones or tablets [40]. It can measure up to 600V and 10A with 24 bit resolution. Voltage and current can be measured simultaneously, with power calculation built into the smartphone application. It also allows for logging to a microSD memory card for up to 6 months.

Chapter 3

System Design

3.1 Energy Harvesting

3.1.1 PV Harvesting

A generic polycrystalline solar panel is used for testing the sensor nodes. The panel has a rated peak power of $0.15W$ and a voltage of $5V$, and physical dimensions of $53x30mm$. The solar panel is depicted in figure 3.1

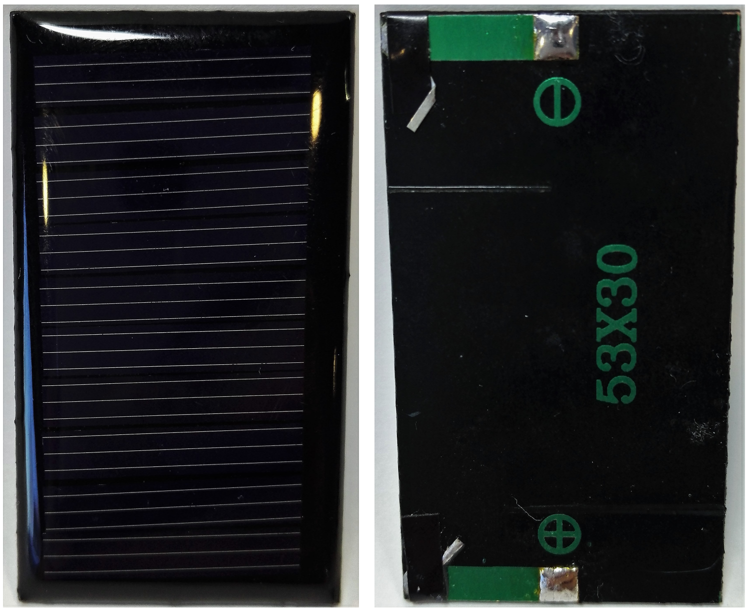


Figure 3.1: $0.15W$ solar panel

In order to obtain the necessary characteristics of the solar panel, a characterization of the panel has been conducted with the Neonsee AAA Sun Simulator at the UiA PV-Lab [41]. Figure 3.2 shows the characterization with $1000W/m^2$ irradiance, equivalent to that received from the sun. Voltage is represented by the x-axis, while current and power are represented by the y-axis.

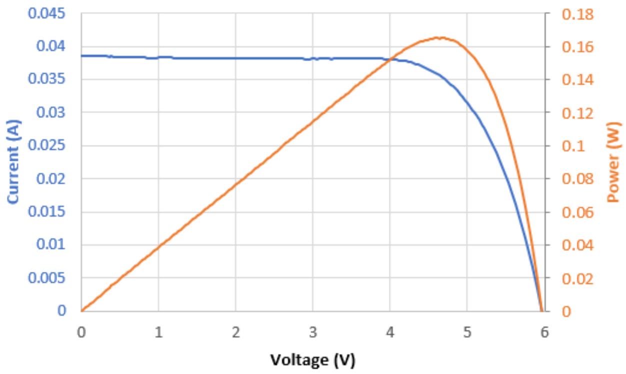


Figure 3.2: Characterization of $0.15W$ rated polycrystalline panel

Should the $0.15W$ panel prove to be insufficient, a larger, generic monocrystalline solar panel is available for use. The panel, which is depicted in figure 3.3 has a rated peak power of $0.36W$ and a voltage of $4V$, and measures $63x63mm$.

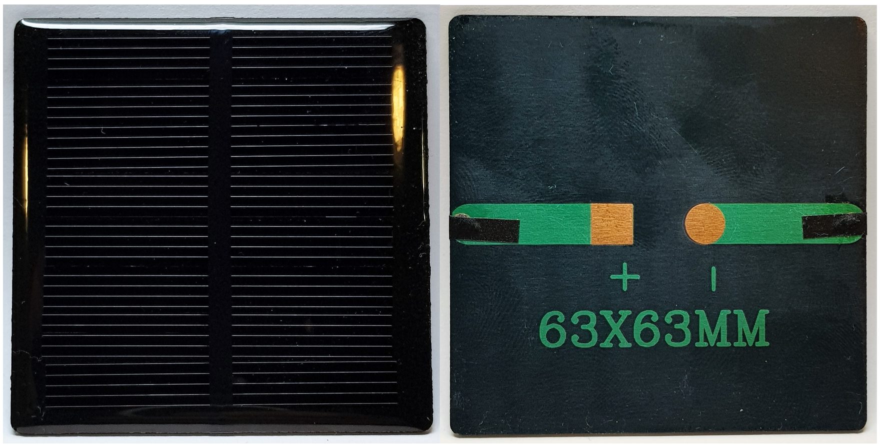


Figure 3.3: Front and back of the $0.36W$ solar panel

Figure 3.4 shows that the panel outputs well above the rated power of $0.36W$.

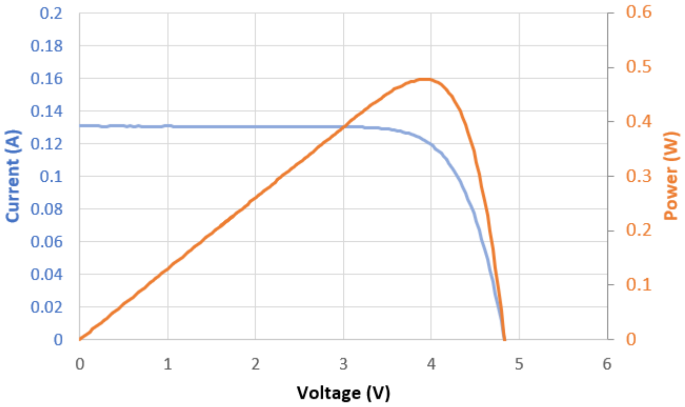


Figure 3.4: Characterization of $0.36W$ rated monocrystalline panel

As can be seen in the figures, the maximum power point is at approximately 80%, as described in section 2.1.1. The MPPT is therefore designed according to these values.

3.1.2 Power Management

Figure 3.5 depicts the power management circuit using the BQ25570 IC, which has features such as MPPT, a Buck/Boost converter for charging lithium ion cells with a PV input, output voltage conditioning and battery voltage security measures. All of these features are programmable through resistor networks [42]. Additionally, it has a cold start voltage of $330mV$ and a quiescent current of $900nA$, making it highly suitable for a solar panel in an indoor environment. Solar panels with voltages up to $5.1V$ and power output of up to $510mW$ can be used with the BQ25570. The circuit depicted is from the ATmega328p node, and the designators of the components may differ from the nRF52840 node.

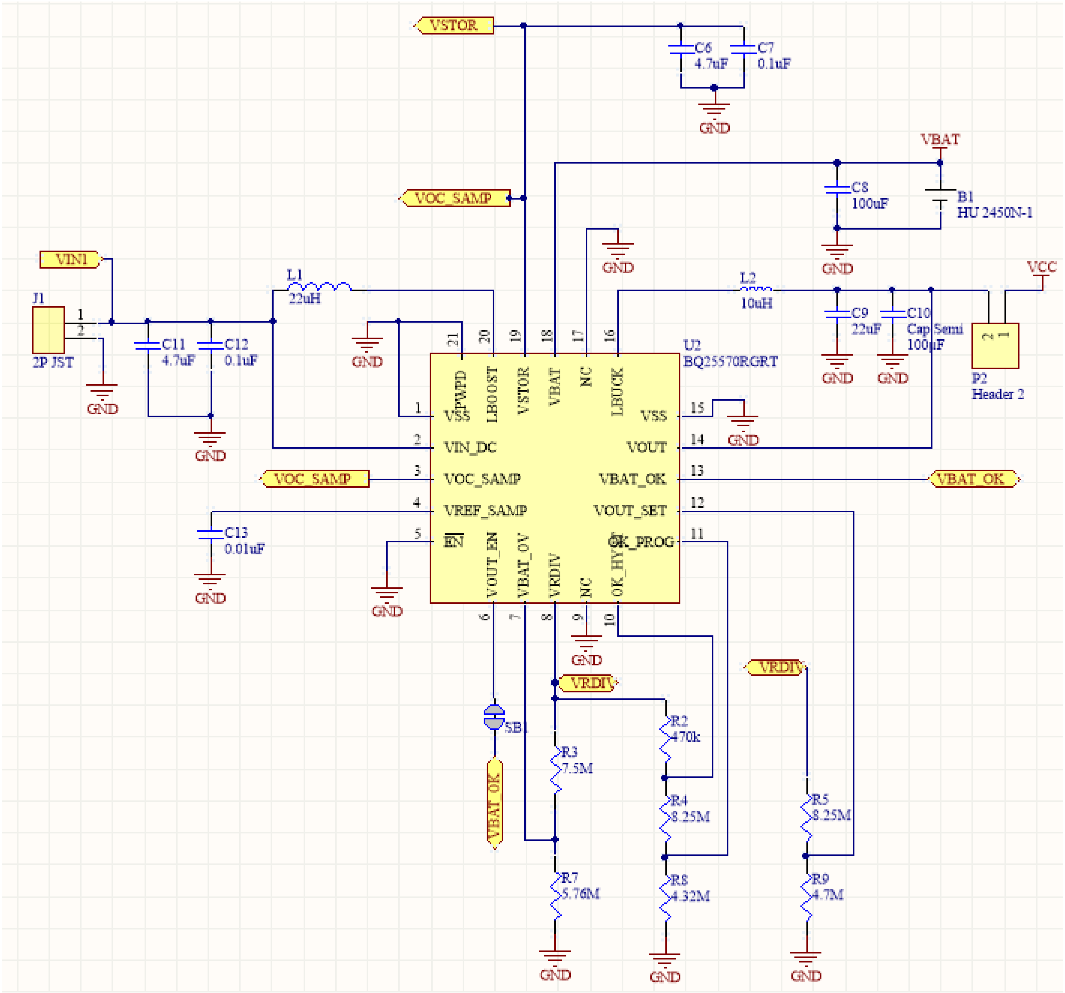


Figure 3.5: BQ25570 power management schematic

HU 2450N-1, B1 in figure 3.5, is a battery holder for the rechargeable lithium-ion battery LIR2450. This is a small $120mAh$ lithium-ion coin cell battery with a nominal voltage of

$3.6V$ meaning it can supply a total of approximately $432mWh$ from fully charged at $4.2V$ to fully discharged at $2.75V$ as specified by the datasheet [11].

The BQ25570 is efficient in both battery charging and output voltage regulation. Figure 3.6 shows the efficiency with a $2V$ input voltage from the solar panel at different input current levels, while figure 3.7 shows the efficiency of the output buck converter at different output current levels with an $1.8V$ output. The output voltage is programmed to $3.3V$ when used in the sensor nodes.

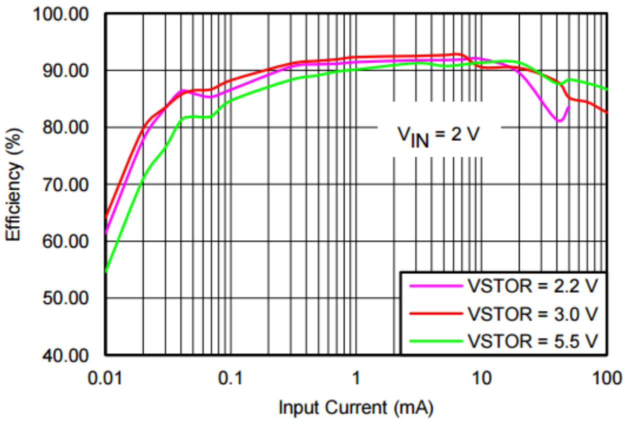


Figure 3.6: BQ25570 charging efficiency vs input current with 2V input

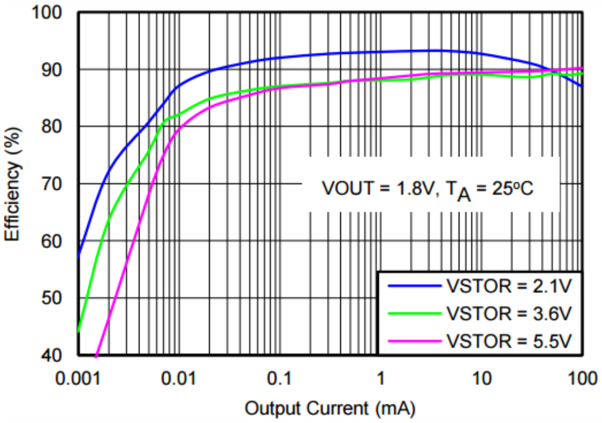


Figure 3.7: BQ25570 buck efficiency vs output current with 1.8V output

Following the theory presented in section 2.1, the MPPT is set to 80%, which is obtained by connecting the VSTOR pin to the VOC_SAMP pin.

3.1.2.1 Battery Overvoltage Protection

In order to prevent the battery from being overcharged, the resistors R3 and R7 are used as a voltage divider to program the BQ25570. According to the datasheet, the sum of the resistors should be as close to $13M\Omega$ as possible [42]. The datasheet specifies $R3 = 7.5M\Omega$ and $R7 = 5.76M\Omega$, giving a sum of $R3 + R7 = 13.26M\Omega$. Equation 3.1 is given by the datasheet to determine the maximum battery voltage.

$$VBAT\_OV = \frac{3}{2} \cdot VBIAS \left(1 + \frac{R3}{R7}\right) \tag{3.1}$$

Where:

- VBIAS is the internal reference for the programmable voltage thresholds, ranging from $1.205V$ to $1.217V$, typically $1.21V$
- R7 is $5.76M\Omega$
- R3 is $7.5M\Omega$

With ideal component values, the maximum charge voltage is given by equation 3.2.

$$VBAT\_OV = \frac{3}{2} \cdot 1.21V \left(1 + \frac{7.5M\Omega}{5.76M\Omega}\right) = 4.18V \tag{3.2}$$

However, considering the VBIAS voltage range as well as resistor tolerances of 1%, the VBAT_OV value can range from $4.16V$ to $4.21V$. These values are within the maximum voltage specifications of a lithium-ion battery.

3.1.2.2 Battery Voltage Threshold

The BQ25570 includes a VBAT_OK pin which can be used by an MCU or switch to shut down when the battery is discharged below a set threshold. Two pins are included to program threshold values, namely the VBAT_OK_PROG, which sets the voltage threshold while discharging, and VBAT_OK_HYST, which sets the threshold while charging. R2, R4 and R8 are used as voltage dividers in order to program the threshold values. The datasheet states that the sum of the resistors should be as close to $13M\Omega$ as possible. Equations 3.3 and 3.4 are given by the datasheet to calculate the threshold voltages.

$$VBAT\_OK\_PROG = VBIAS \left(1 + \frac{R4}{R8}\right) \tag{3.3}$$

$$VBAT\_OK\_HYST = VBIAS \left(1 + \frac{R4 + R2}{R8}\right) \tag{3.4}$$

The BQ25570 output voltage buck converter has a maximum dropout voltage of $0.2V$. Therefore, VBAT_OK_PROG should be minimum $3.5V$ to safely output $3.3V$ to the circuit. The VBAT_OK_HYST should be high enough to allow sufficient charge for transmitting data. Based on this, the following values are used:

- VBIAS is the internal reference for the programmable voltage thresholds, ranging from $1.205V$ to $1.217V$, typically $1.21V$
- R2 is $470k\Omega$
- R4 is $8.25M\Omega$
- R8 is $4.32M\Omega$

With ideal component values, the VBAT_OK thresholds are given by equations 3.5 and 3.6.

$$VBAT\_OK\_PROG = 1.21V \left(1 + \frac{8.25M\Omega}{4.32M\Omega}\right) = 3.52V \tag{3.5}$$

$$VBAT\_OK\_HYST = 1.21V \left(1 + \frac{8.25M\Omega + 470k\Omega}{4.32M\Omega}\right) = 3.65V \tag{3.6}$$

The VBAT_OK pin will be high until the battery voltage goes below $3.521V$ while discharging, which will set the pin low. When the battery is recharged to $3.65V$, the pin returns to a high state. The VBAT_OK pin is connected to the VOUT_EN pin, which enables or disables the output of the buck converter supplying the node with power. By doing this, the system is ensured to be supplied with the correct voltage at all times.

3.1.2.3 Output Voltage

The datasheet gives equation 3.7 to calculate the output voltage of the BQ25570. The example circuit is however designed for $1.8V$ output, and must therefore be recalculated.

$$VOUT = VBIAS\left(\frac{R5 + R9}{R9}\right) \tag{3.7}$$

Where:

- VBIAS is the internal reference for the programmable voltage thresholds, ranging from $1.205V$ to $1.217V$, typically $1.21V$
- R5 is unknown
- R9 is unknown

R5 and R9 are used as a voltage divider to give the correct voltage to the VOUT_SET pin. In order to find the resistor values for the required $3.3V$ output, equation 3.7 is rearranged as shown in equation 3.8, where the specified sum of $R5 + R9 = 13M\Omega$ is used.

$$R9 = \frac{VBIAS(R5 + R9)}{VOUT} = \frac{1.21V \cdot 13M\Omega}{3.3V} = 4.77M\Omega \tag{3.8}$$

R5 is then found to be $R5 = 13M\Omega - 4.77M\Omega = 8.23M\Omega$. The closest available values are $R5 = 8.2M\Omega$ and $R9 = 4.7M\Omega$, resulting in $VOUT = 3.32V$

3.1.3 $1.8V$ Operation

The nRF52840 supports $1.8V$ operation, further reducing the energy consumption of the node. As mentioned in section 1.3.2, this will be implemented and tested if time permits. With $1.8V$ operation, more of the stored energy in the battery can be utilized with modifications of the circuit.

The output voltage is reduced to $1.8V$ by using the resistor values proposed in the BQ25570 datasheet, where $R5 = 4.22M\Omega$, and $R9 = 8.66M\Omega$. This results in $VOUT = 1.8V$. These resistor values are, however, not available at UiA at the time of assembly. Therefore, the $VOUT$ is programmed to $1.9V$ by using $R5 = 4.7M\Omega$ and $R9 = 8.25M\Omega$. resulting in $VOUT = 1.9V$

As previously stated, the LIR2450 battery can operate safely down to $2.75V$. In order to reduce the battery threshold voltages to utilize more of the available energy, VBAT_OK_-PROG and VBAT_OK_HYST are modified. By using $R2 = 887k\Omega$, $R4 = 6.98\Omega$ and $R8 = 5.36M\Omega$, the resulting values are calculated in equations 3.9 and 3.10.

$$VBAT\_OK\_PROG_{1.8V} = 1.21V\left(1 + \frac{6.98M\Omega}{5.36M\Omega}\right) = 2.79V \tag{3.9}$$

$$VBAT\_OK\_HYST_{1.8V} = 1.21V\left(1 + \frac{6.98M\Omega + 887k\Omega}{5.36M\Omega}\right) = 2.99V \tag{3.10}$$

3.1.4 Operation with Supercapacitor

A design for operation with supercapacitors has been developed. In order to obtain the highest amount of energy with any given capacitance value, supercapacitors with a voltage rating of $6V$ are used, as the BQ25570 has a maximum charging voltage of $5.5V$. The resistor networks are modified to better suit the higher voltages. The BQ25570 datasheet specifies that the initial charge time may be significant with a large, depleted supercapacitor.

The maximum charge voltage is modified by changing the resistor values of $R3$ and $R7$ to $9.31M\Omega$ and $3.74M\Omega$ respectively. This results in $R3 + R7 = 13.05M\Omega$, which is sufficiently close to the $13M\Omega$ specified by the BQ25570 datasheet. Equation 3.11 shows the calculated maximum charge voltage.

$$VBAT\_OV_{supercap} = \frac{3}{2} \cdot 1.21V\left(1 + \frac{8.25M\Omega}{4.32M\Omega}\right) = 5.28V \tag{3.11}$$

The resistor network for VBAT_OK_PROG and VBAT_OK_HYST are modified to operate at $3.64V$ and $4.11V$ respectively. The increased hysteresis voltage is modified to ensure that the supercapacitor has sufficient energy for multiple transmissions. These voltages are obtained by using the resistor values $R2 = 1.5M\Omega$, $R4 = 7.68M\Omega$ and $R8 = 3.83M\Omega$, as shown in equations 3.12 and 3.13. The resistor values result in $R2 + R4 + R8 = 13.01M\Omega$, which is sufficiently close to the $13M\Omega$ specified by the BQ25570 datasheet.

$$VBAT\_OK\_PROG_{supercap} = 1.21V\left(1 + \frac{7.68M\Omega}{3.83M\Omega}\right) = 3.64V \tag{3.12}$$

$$VBAT\_OK\_HYST_{supercap} = 1.21V\left(1 + \frac{7.68M\Omega + 1.5M\Omega}{3.83M\Omega}\right) = 4.11V \tag{3.13}$$

The amount of energy in a supercapacitor in joules is given by equation 3.14

$$E = \frac{1}{2} \cdot (V_{max} - V_{min})^2 \cdot C \tag{3.14}$$

Where:

- E is the energy in joules
- V_{max} is the maximum voltage to which the supercapacitor will charge
- V_{min} is the minimum voltage to which the supercapacitor will discharge
- C is the capacitance of the supercapacitor in farads

By assuming the above values, as well as a $1.5F$ supercapacitor which is tested during the project, the total stored energy is found in equation 3.15

$$E = \frac{1}{2} \cdot (5.28V - 3.64V)^2 \cdot 1.5F = 2.02J \tag{3.15}$$

This corresponds to $2.02J \cdot (1/3600s) = 560\mu Wh$. Table 3.1 shows the amount of available power with different capacitance values and for both $1.8V$ and $3.3V$ operation based on the maximum voltage of supercapacitor voltage of $5.28V$. For $1.9V$, a VBAT_OK_PROG value of $2V$ is used, while the calculated $3.64V$ is used for $3.3V$ operation. $6V$ supercapacitors from AVX are used, as they provide a low self discharge current compared to other providers.

Table 3.1: Supercapacitor and voltage comparison

Capacitance	**1.9V Operation**	**3.3V Operation**	**Leakage**
1.5F	$2.24mWh$	$560\mu Wh$	$18\mu A$
2.5F	$3.74mWh$	$934\mu Wh$	$30\mu A$
5F	$7.47mWh$	$1.87mWh$	$36\mu A$

In comparison, the LIR2450 battery has a total capacity of $120mAh \cdot 3.6V = 432mWh$ in optimal conditions.

3.2 Sensors

Various sensors have been researched, evaluated and tested throughout the project. The finalized nodes mainly use the BME680, described in section 3.2.1, because of its features of temperature, RH, IAQ and atmospheric pressure sensing. Support for the HDC2010, described in section 3.2.2, is added in both firmware and hardware because of their lower price and power consumption in case less environmental values are required. Additionally, the ULP MAX44009, which is described in section 3.2.3, is used to sense visible light levels.

Figure 3.8 shows an overview of the supported sensors along with information on which environmental values they can sense.

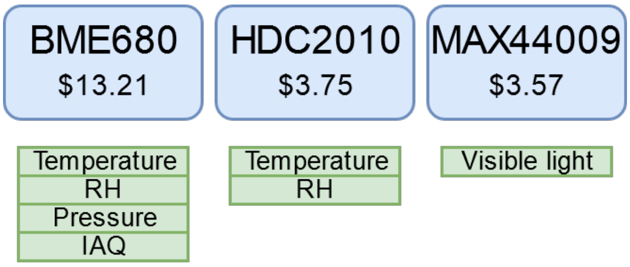


Figure 3.8: Overview of sensors supported by the nodes. Prices are for one sensor on Digikey 15.03.2018

3.2.1 BME680

The BME680 is an integrated environmental sensor developed by Bosch Sensortec [43]. It can be used to measure temperature, RH, atmospheric pressure and IAQ, and supports communication via both I^{2}C and SPI. With readings of temperature and RH every $1s$, the BME680 consumes $3.7\mu A$ average as specified by the datasheet. Measuring IAQ consumes much more power, as the heater has a current consumption of maximum $15mA$. In many cases, IAQ and atmospheric pressure measurements are superfluous, in which case the cheaper HDC1010 can be used instead.

Apart from the high cost of the BME680, it is excellent for use in ULP sensor nodes due to its extremely low current consumption. The key specifications from the datasheet are listed in table 3.2 [43].

Table 3.2: Key features of BME680

Parameter	**Symbol**	**Typical Value**	**Max Value**
Supply Voltage	V_{DD}	$1.8V$	$3.6V$
Sleep Current	I_{DDSL}	$0.15\mu A$	$1\mu A$
Standby Current	I_{DDSB}	$0.29\mu A$	$0.8\mu A$
Start-up Time	$t_{startup}$		$2ms$
Dimensions	$B \cdot D \cdot H$	$3.0 \cdot 3.0 \cdot 0.93mm$	
Slave address		0x77	
Measurement Current	**Symbol**	**Typical Value**	**Max Value**
RH	I_{DDH}	$340\mu A$	$450\mu A$
Pressure	I_{DDP}	$714\mu A$	$849\mu A$
Temperature	I_{DDT}	$350\mu A$	
Air Quality Heater	I_{DD}	$12mA$	$15mA$
Range	**Symbol**	**Min Value**	**Max Value**
RH	H_A	0%	100%
Pressure	P	$300hPa$	$1100hPa$
Temperature	T_A	$-40°C$	$85°C$
Air Quality	IAQ_{rg}	0	500
Accuracy	**Symbol**	**Typical Value**	**Max Value**
RH	A_H	±3%	
Pressure (300-1100hPa)	$A_{p,full}$	$\pm 0.6hPa$	
Temperature (0-65°C)	$A_{T,full}$	$\pm 1°C$	
Air Quality	AX_{IAQ}	±15%	

Table 3.3 shows the addresses of the registers used to configure and read sensor data from the BME680.

Table 3.3: BME680 I^{2}C registers

Name	Address
Measurement Status	0x1D
Pressure	0x1F - 0x21
Temperature	0x22 - 0x24
Humidity	0x25 - 0x26
Gas	0x2A - 0x2B
Config	0x75
Calibration Reg1	0x89
Chip ID	0xD0
Reset	0xE0
Calibration Reg2	0xE1

3.2.2 HDC2010

The HDC2010 from Texas Instruments is used to sense both temperature and RH [44]. With its ULP friendly energy consumption and sufficiently accurate readings, it is suitable to the WSN in every aspect. The HDC2010 uses I^{2}C for communication with the MCU.

The RH sensor is vulnerable to being offset by high temperatures or long exposures of high RH. To assist in removing the offset, the HDC2010 has an internal heater that increases the temperature, effectively removing unwanted condensation. This is, however, not implemented in the design, as this is not likely to occur in an indoor environment, and would consume more energy. Table 3.4 outlines the relevant characteristics of the HDC2010.

Table 3.4: HDC2010 characteristics

Parameter	**Value**	**Unit**
Supply voltage	1.62 – 3.6	V
Sleep	50	nA
Average @ 1 measurement/second, RH(11bit) + temp(11 bit)	550	nA
Temperature accuracy	± 0.2	$^\circ C$
RH Accuracy	± 2	$\%RH$
Temperature conversion time (9/11/14 bit)	225/350/610	μs
RH conversion time (9/11/14 bit)	275/400/660	μs
Dimensions	$1.5 \cdot 1.5 \cdot 0.675$	mm
Slave address		0x41

Table 3.5 shows the addresses of the registers used to configure and read sensor data from the HDC2010.

Table 3.5: HDC2010 registers

Name	**Address**
Temperature	0x00 - 0x01
Humidity	0x02 - 0x03
Configuration Register	0x0E
Measurement Configuration Register	0x0F

The configuration register with the address 0X0E has 8 bits which can be used to configure the HDC2010. The different bits, with their functions, are outlined in table 3.6

Table 3.6: HDC2010 configuration register

BIT	**Description**
7	Reset
6:4	Data rate
3	Heater
2	Interrupt
1	Interrupt polarity
0	Interrupt mode

In addition to the configuration register, a measurement configuration register of 8 bits is located at 0X0F. The bits, with their functions, are outlined in table 3.7.

Table 3.7: HDC2010 measurement configuration register

BIT	**Description**
7:6	Temperature resolution
5:4	Humidity resolution
3	Reserved
2:1	Measurement configuration
0	Measurement trigger

3.2.3 MAX44009

The MAX44009 from Maxim Integrated is used to sense ambient light levels. It can sense between the range of $0.045-188000LUX$ with 22-bit dynamic range, making it ideal for both indoor and outdoor use. It uses I^{2}C for communication with the MCU. Its on-chip photodiode is optimized to mimic the ambient light perception of the human eye, and incorporates Infrared Radiation (IR) and Ultraviolet (UV) blocking capability.

Table 3.8: MAX44009 characteristics

Parameter	Value	Unit
Supply voltage	1.7 – 3.6	V
Minimum signal integration time	6.25	ms
Continuous current consumption	650	nA
Total error	15	%
Dimensions	$2 \cdot 2 \cdot 0.6$	mm
Slave address		0x4b

Table 3.9 shows the addresses of the registers used to configure and read sensor data from the MAX44009.

Table 3.9: MAX44009 registers

Name	Address
Configuration register	0x02
Lux	0x03-0x04

The configuration register located at 0x02 contains 8 bits for configuring the MAX44009. The bits and their functions are described in table 3.10.

Table 3.10: MAX44009 configuration register

BIT	Description
7	Continuous mode
6	Manual mode
5:4	Reserved
3	Current division ratio
2:0	Integration time

3.2.4 Battery Level Sensing

The sensor nodes are supplied with $1.9V$ or $3.3V$, while a Li-Ion battery has a maximum charge of $4.2V$, and a supercapacitor charged by the BQ25570 has a maximum charge of $5.281V$. Therefore, in order to measure and monitor the battery voltage level without damaging the MCU or other connected devices, a voltage divider is required to obtain tolerable values for the Analog-to-Digital Converter (ADC). A voltage divider does, however, consume high amounts of constant energy relative to the requirements of the sensor nodes. Therefore, an ultra low leakage switch, the TPS22860, is implemented to turn the supplied battery voltage on only when taking measurements, and otherwise off [5]. The TPS22860 has a maximum V_{in} leakage current of $50nA$, and a maximum V_{bias} leakage current of $100nA$ at $3.3V$, and supports loads of up to $200mA$.

Figure 3.9 shows the schematic of the battery level sensing. The capacitors are the recommended value from the TPS22860 datasheet. When the battery level is to be read by an ADC, the "batSensorSwitch" signal is switched to HIGH by the MCU, causing the switch to close and allow the "VBAT" signal, which is connected directly to the battery, to pass through to the Vout pin of the TPS22860. The voltage divider then halves the voltage to the ADC, which is the "BatLvl" signal, following equation 3.16.

$$BatLvl = Vout \cdot \frac{R10}{R6 + R10} = Vout \cdot \frac{10k\Omega}{10k\Omega + 10k\Omega} = \frac{1}{2} \cdot Vout \tag{3.16}$$

The current consumption of the voltage divider when the switch is on is at most $I_{Vdiv} = (4.2V)/(10k\Omega + 10k\Omega) = 210\mu A$

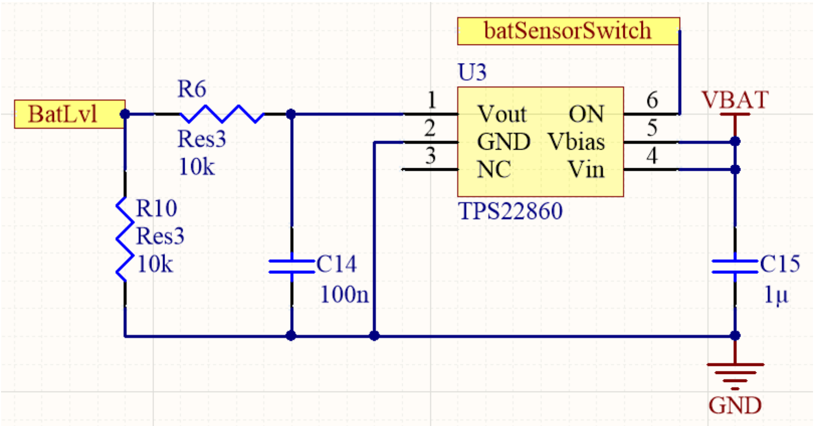


Figure 3.9: Battery level sensing schematic

For the nRF52840, the ADC voltage reference used is $0.6V$. Therefore, the "BatLvl" signal should never be higher than $0.6V$. In order to facilitate for the possible maximum voltages of $4.2V$ and $5.281V$, the voltage must be divided by a factor of at least 8.8. Changing the value of R6 to the commonly used value of $82k\Omega$, while keeping R10 at $10k\Omega$, ensures a maximum "BatLvl" voltage of $0.574V$. When using $R6 = 82k\Omega$, the maximum current consumption is $I_{Vdiv} = (5.281V)/(82k\Omega + 10k\Omega) = 57.4\mu A$.

3.3 WSN Communication Protocols

Two different communication protocols have been designed, one for multi-hop topology, allowing communication between nodes, and one for star topology with communication directly between nodes and a gateway. The designs include mechanisms to detect and correct failed transmissions, avoid collision and adjust transmission power dynamically.

3.3.1 Medium Access Mechanisms

Failed Transmission Detection

ACK messages are implemented to ensure that transmitted messages are received. This allows for retransmission of broken or lost packages. It improves the reliability of the network at the cost of a small increase in power consumption, since the transceiver and microcontroller on both ends have to be awake for an increased amount of time. This mechanism can be deactivated for applications where packet loss is of less importance.

Collision Avoidance

Each retransmission of a package consumes extra power. In order to reduce failed transmissions due to collisions, the functionality for random backoff and carrier sensing is implemented. The random backoff functionality is particularly useful for the multi-hop protocol described in 3.3.3, as data transmissions are initiated by a broadcast message. When receiving this broadcast, all nodes which have data to be transmitted waits for a short random interval before performing a carrier sensing to check if the medium is idle or not. The node that drew the shortest interval will be able to transmit its data, while other nodes adjacent to the relaying node need to wait for the next round. As for the one-hop protocol in star topology, only carrier sensing is used, as transmissions are not controlled by an external event and may happen at any time.

Power Level Adjustment

Transmitting data with higher power level than necessary consumes excessive amounts of power and creates more interference for other devices. Dynamic power level adjustment mechanisms have been implemented for both the ATmega328p node and the nRF52840 node in order to automatically change the power level based on the range and noise in the channel. Due to the two transceivers having different properties, two different mechanisms are required, where both rely on the use of ACK messages. For the purpose of this project, dynamic power level adjustment is implemented only in star topology due to the increased complexity of the multi-hop protocol.

The mechanism for the nRF52840 node is based on a Received Signal Strength Indicator (RSSI). When a node transmits a data package, the receiver senses RSSI and adds it as a payload in the ACK package. Additionally, the node senses RSSI while receiving the ACK, and adjusts transmit power according to the lowest value of the two RSSI values and sensitivity in the receivers. The power level is also included in each data package so that the receiver can return an ACK with equal power.

The nRF24l01+PA+LNA transceiver on the ATmega328p node can not measure RSSI, which limits the possibilities of dynamically adjusting the power level. Therefore, the power level adjustment is based on packet loss. If a node attempts transmission a given number of times without receiving an ACK, it enters a power adjustment phase. The power level is set to the lowest value, and incremented per given number of failed transmissions until an ACK is received. This is a costly process with regards to power consumption, and a relatively high number of unsuccessful transmissions should be accepted before entering this phase.

3.3.2 Star Protocol

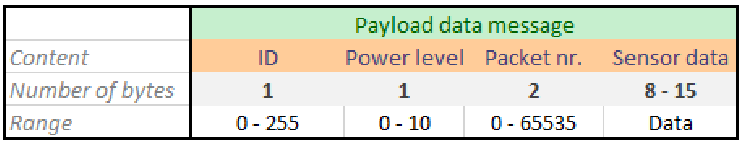
Star topology has a low level of complexity as a gateway is always on and listening on the channel, and each node communicates directly with the gateway. Therefore, surrounding nodes are able to wake up, gather data and transmit the data to the gateway in a short amount of time. Despite the short transmission times, the mechanism for collision avoidance by channel sensing has been implemented in order to further reduce the risk of failed transmissions due to collision. Additionally, the protocol utilizes ACK messages to ensure reliable data gathering, as well as power level adjustment for minimizing energy consumption and interference on the channel.

The steps to the communication protocol for star topology are as follows:

- Gateway is always on and listening for messages
- The node which has data to send performs carrier sensing
- If the channel is busy, the node will wait until it is idle
- If or when the channel is sensed as idle, the node will transmit its data message to the gateway and start listening for an ACK message
- The gateway returns an ACK message with the same power level
- If the node does not receive ACK, it will return to the second step and attempt retransmission of the original message

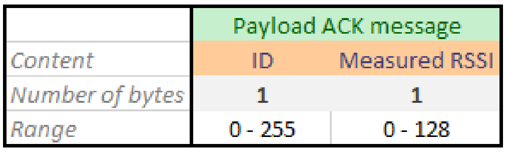
- If the node receives ACK, the transmission is considered finished and the node can return to sleep.

Figure 3.10 shows the payload structure of the data messages with data from all sensors grouped in field *Sensor data*, and figure 3.11 the payload of the ACK messages used in the star protocol for the nRF52840 WSN.



	Payload data message			
Content	ID	Power level	Packet nr.	Sensor data
Number of bytes	1	1	2	8 - 15
Range	0 - 255	0 - 10	0 - 65535	Data

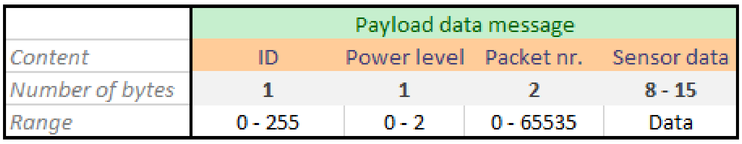
Figure 3.10: Payload of data message with nRF52840 in star



	Payload ACK message	
Content	ID	Measured RSSI
Number of bytes	1	1
Range	0 - 255	0 - 128

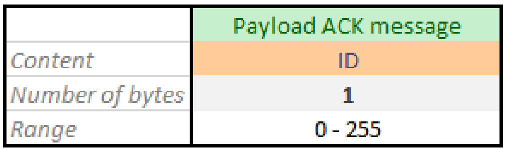
Figure 3.11: Payload of ACK message with nRF52840 in star

Figure 3.12 shows the payload structure of the data messages with data from all sensors grouped in field *Sensor data*, and figure 3.13 the payload of the ACK messages used in the star protocol for the ATmega328p WSN.



	Payload data message			
Content	ID	Power level	Packet nr.	Sensor data
Number of bytes	1	1	2	8 - 15
Range	0 - 255	0 - 2	0 - 65535	Data

Figure 3.12: Payload of data message with ATmega328p in star



	Payload ACK message
Content	ID
Number of bytes	1
Range	0 - 255

Figure 3.13: Payload of ACK message with ATmega328p in star

3.3.3 Multi-hop Protocol

Multi-hop topology enables extended range with less infrastructure. BT5 supports mesh topology, but due to topology requirements mentioned in section 2.3.1, it can not run on a network of battery powered nodes with limited energy harvesting. Therefore, a protocol that enables multi-hop communication for both the ATmega328p node and nRF52840 node has been developed.

With this protocol, a network of nodes is able to route messages towards a dedicated gateway, which forwards the received data to an MQTT broker via Wi-Fi. The protocol is designed so that only the gateway needs to be constantly on and awake. Implementation of the protocol somewhat differs between the two different nodes as the transceivers have different abilities, however the core functionality remains the same. Implementation for both the ATmega328p node and nRF52840 node is described in section 3.4.2.10 and 3.6.2.9 respectively.

Figure 3.14 shows the basic concept of the nodes in the multi-hop network. At startup of each node, the radio is initialized with the necessary registry modifications. A timer is then initiated, and the node enters sleep state for a given interval of time. After this interval, the node wakes up and checks the state of the timer. If less than a programmed period of time has passed, the node broadcasts an wake ping message, and waits for replies from nodes that want to relay data via itself. If no neighboring node wants to relay data, the node goes back to sleep. If a neighboring node transmits data, the node forwards this data to the first node that transmits an wake ping message with number of hops to gateway lower than that of the relaying node. The node also updates its hops to gateway equal to that of the next hop node plus one before going back to sleep. If the timer is above the set level, the node will attempt transmission of its own sensor data in the same manner as when relaying data.

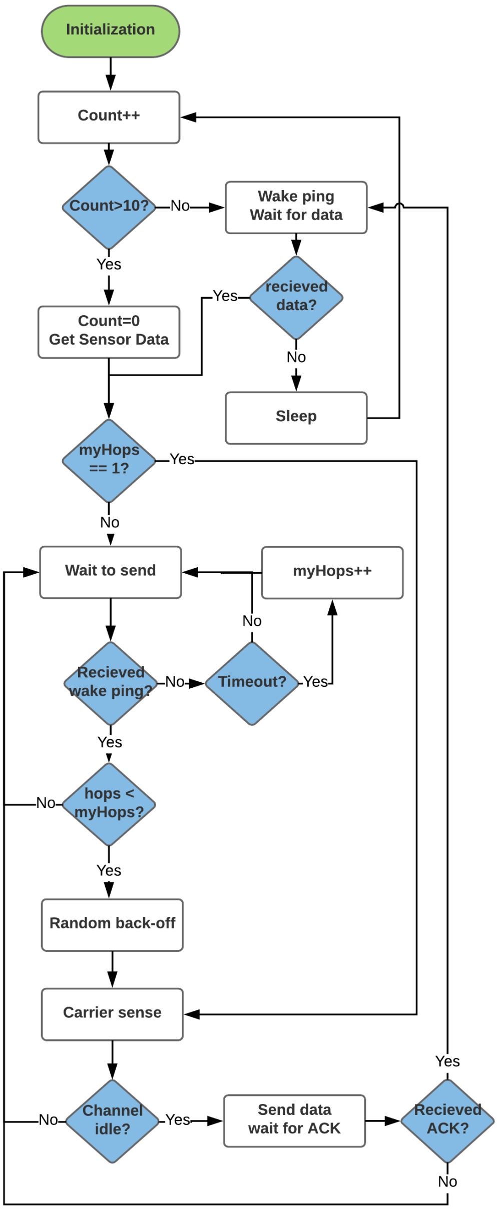


Figure 3.14: Multi-hop protocol flowchart

The gateway(s) broadcast wake ping messages with hops to gateway set to zero. Adjacent nodes identifies this as a gateway node, and sets their hops to gateway to one. Routes to the gateway will then propagate throughout the network. As any node that broadcasts hops to gateway = 0 is identified as a gateway, several gateways can be placed in the same network, and surrounding nodes will simply transmit towards the nearest one in terms of hops.

Figure 3.15 depicts a scenario where 5 nodes are communicating with the multi-hop protocol. The yellow nodes in tile 1 and 2 want to transmit data towards gateway, and waits for a wake ping message with hops to gateway lower than their own. As can be seen in the first and second tile, the wake ping message with hops equal to three is ignored by both the yellow nodes as they have previously used a route with 2 and 3 hops to gateway. In tile 2, both yellow nodes receive a wake ping message with hops to gateway equal to one. To minimize the risk of collisions, both nodes wait for a short random amount of time before performing a carrier sensing. The node that selects the shortest waiting interval senses the channel as idle, and starts to transmit its data. Upon successful transmission, it will also update its hops to gateway to 2, as it has discovered a shorter route. The other node lost its chance to transmit and waits for the next wake ping message. The green node in tile 6 sends a new wake ping message after it has finished forwarding the previous message. This behaviour continues until there are no more messages to forward. It will then return to sleep.

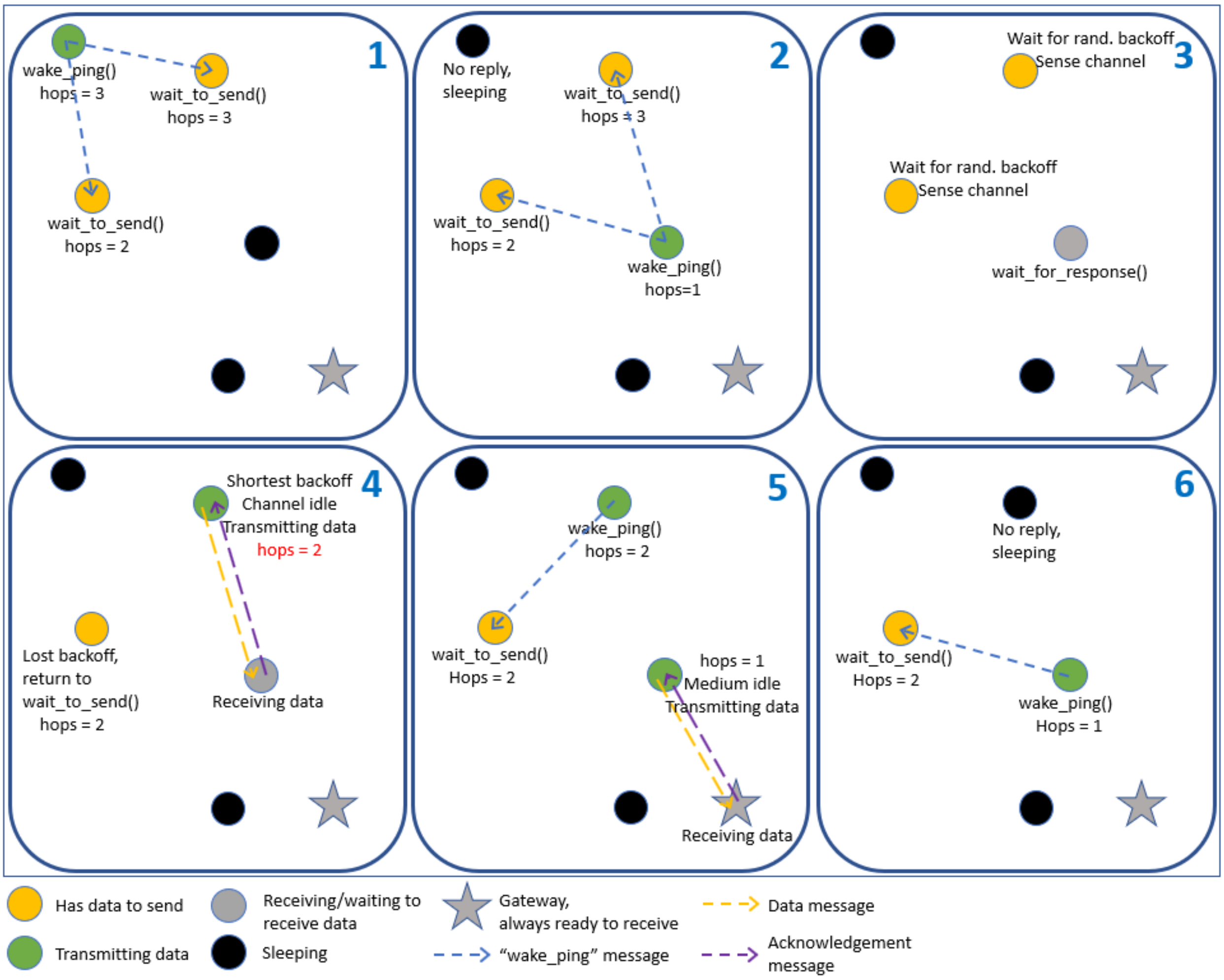
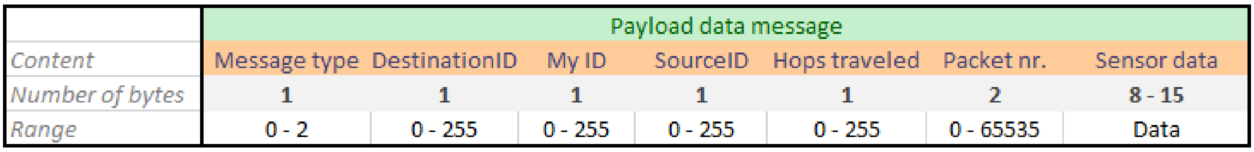


Figure 3.15: multi-hop protocol scenario

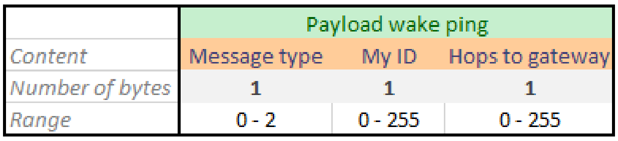
Figure 3.16 shows the payload of the data message used in the multi-hop protocol.



	Payload data message						
Content	Message type	DestinationID	My ID	SourceID	Hops traveled	Packet nr.	Sensor data
Number of bytes	**1**	**1**	**1**	**1**	**1**	**2**	**8 - 15**
Range	0 - 2	0 - 255	0 - 255	0 - 255	0 - 255	0 - 65535	Data

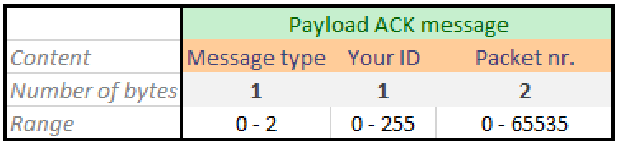
Figure 3.16: Payload of data message

Figure 3.17 depicts the payload of the wake ping broadcast message, while figure 3.18 shows the payload of the ACK message as implemented in the multi-hop topology.



	Payload wake ping		
Content	Message type	My ID	Hops to gateway
Number of bytes	**1**	**1**	**1**
Range	0 - 2	0 - 255	0 - 255

Figure 3.17: Payload of wake ping message



	Payload ACK message		
Content	Message type	Your ID	Packet nr.
Number of bytes	**1**	**1**	**2**
Range	0 - 2	0 - 255	0 - 65535

Figure 3.18: Payload of ACK message

3.4 ATmega328p Node

3.4.1 ATmega328p Hardware

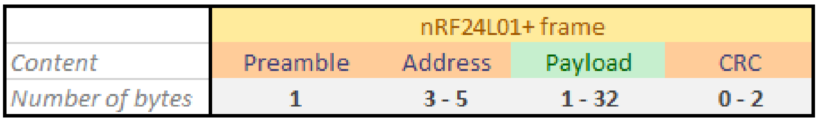
The ATmega328p is a low-cost MCU with support for the Arduino bootloader, making rapid prototyping possible. Additionally, by using Arduino, readymade development boards such as the Arduino Nano, which requires $5V$ supply, or Arduino Pro Mini, which can run at $3.3V$, can be used for testing new sensors and features in a simple and efficient manner [45][46]. Most peripherals, such as transceivers and sensors, have libraries written by the Arduino community, which can be installed through the Arduino IDE.

3.4.1.1 Wireless Transceiver for ATmega328p

One of the radios commonly used with Arduino is the low-cost, highly integrated nRF24l01+, which is based on a proprietary protocol and offers modification of all necessary parameters, such as transmission power [47]. It operates in the $2.4GHz$ ISM band. The nRF24l01+ is available in various modules, from tiny, low-range modules with on-board antenna to bigger, long-range modules with external antenna. Based on its simplicity when used with Arduino libraries, alterability and possibility for long ranges, the nRF24l01+ is used. SPI is used for communication between the ATmega328p and nRF24l01+, where specific values can be written to the nRF24l01+ register in order to change its configuration. The nRF24l01+ can be set to four different power levels represented from lowest to highest as; *MIN*, *LOW*, *HIGH* and *MAX*. The range is, as claimed by vendors, up to $100m$ in line-of-sight, however unofficial tests have proven the range to be approximately $30m$. The range is reduced in urban or indoor environments.

Modules based on the nRF24l01+ IC with higher power amplification and external antennas exist. One of these modules, the nRF24l01+PA+LNA, is used for the purpose of this project in order to obtain a higher range. The module is obtained at a very low cost, and therefore lacks a proper datasheet, however tests will be conducted in order to determine its characteristics.

Figure 3.19 depicts the data frame used with nRF24L01+ transceiver modules.



	nRF24L01+ frame			
Content	Preamble	Address	Payload	CRC
Number of bytes	1	3 - 5	1 - 32	0 - 2

Figure 3.19: nRF24L01+ data frame

3.4.1.2 Schematic and PCB Design

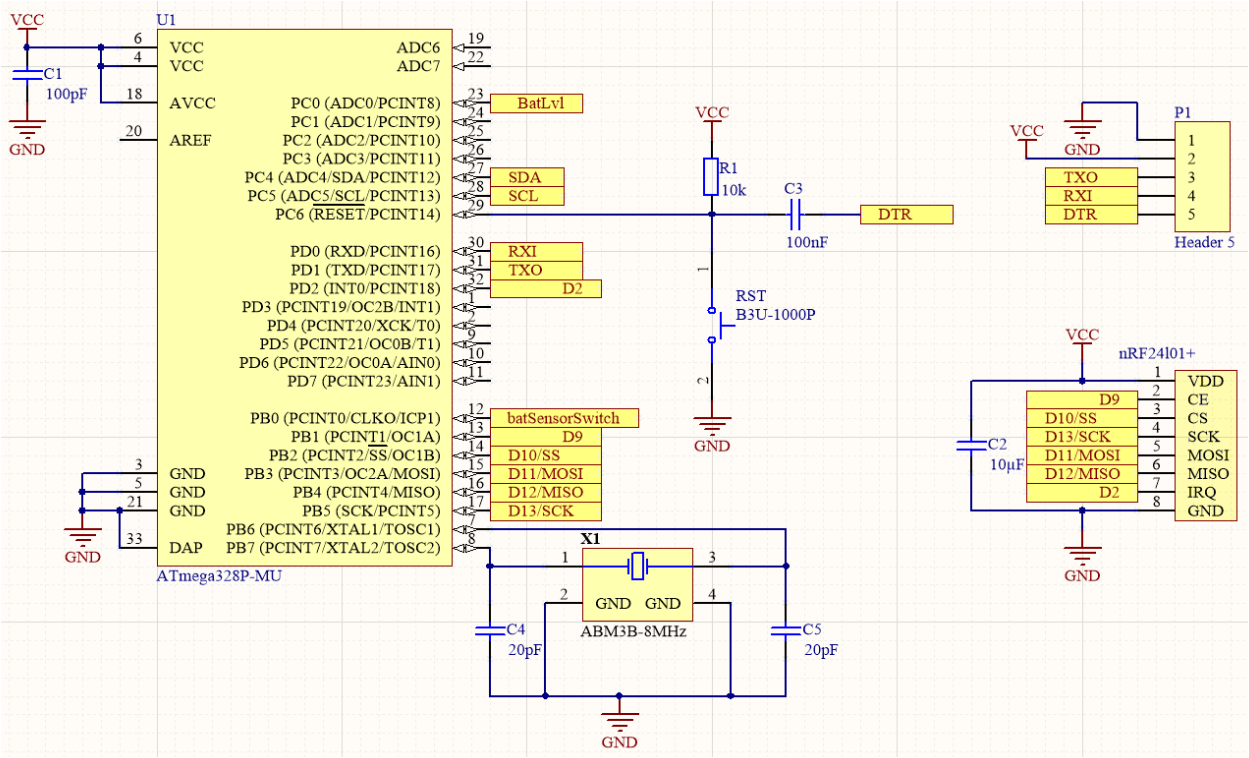


Figure 3.20: Schematic of the ATmega328p

Figure 3.20 depicts part of the final design of the ATmega328p sensornode with the MCU (U1) and its supporting components, as well as the nRF24l01+. In order to flash the Arduino bootloader onto the MCU, the SCK, MOSI and MISO pins are used. Subsequent to this, the firmware can be uploaded directly to the MCU by connecting a USB to UART converter to the "P1" pin header. Additionally, a reset switch (RST) is included for simple resetting of the MCU. All schematics for the ATmega328p node, as well as the PCB, can be found in appendix D.1.

The ATmega328p supports oscillators of up to 20MHz with supply voltages of 4.5V and higher, as shown in figure 3.21, however by reducing the oscillator frequency and supply voltage, the energy consumption is greatly reduced. Figure 3.22 shows a comparison of the power consumption of the ATmega328p with different voltages and oscillator frequencies with a generic, blinking Light Emitting Diode (LED) example code. Based on this, an oscillator of 8MHz (X1) and a supply voltage of 3.3V from the BQ25570 is utilized in the sensornode.

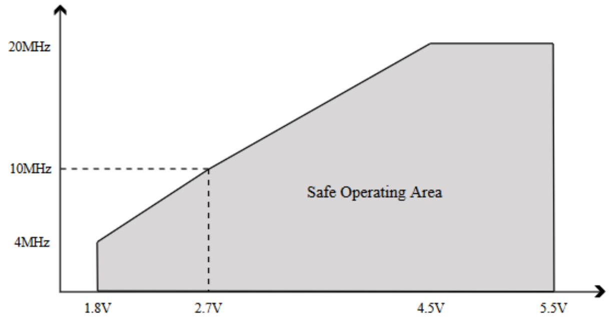


Figure 3.21: Speed grade of the ATmega328p [3]

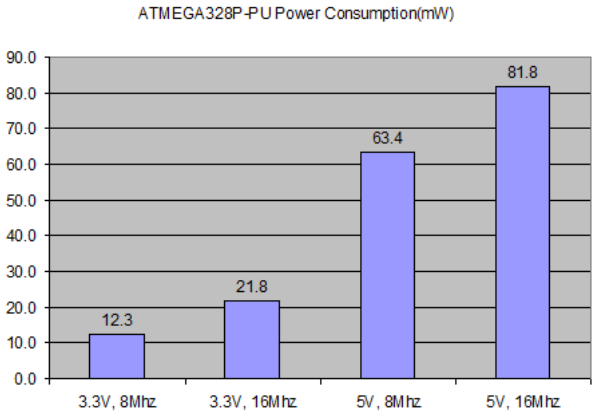


Figure 3.22: Energy consumption at different voltages and frequencies [4]

Figure 3.23 shows the I^{2}C sensors supported by the node. As the BME680 (U7) measures the same values as the HDC2010 (U4), temperature and RH, with the addition of IAQ and atmospheric pressure, only one of them is installed per node depending on the energy requirements and physical placement of the nodes. The MAX44009 (U5) is used for visible light measurements. The sensors are supplied with 3.3V from the buck converter of the BQ25570, and to SCL and SDA on the MCU with pullup resistors on both signal lines (R11, R12). All sensors have decoupling capacitors (C16, C18, C19) from VCC to ground in order to shunt noise.

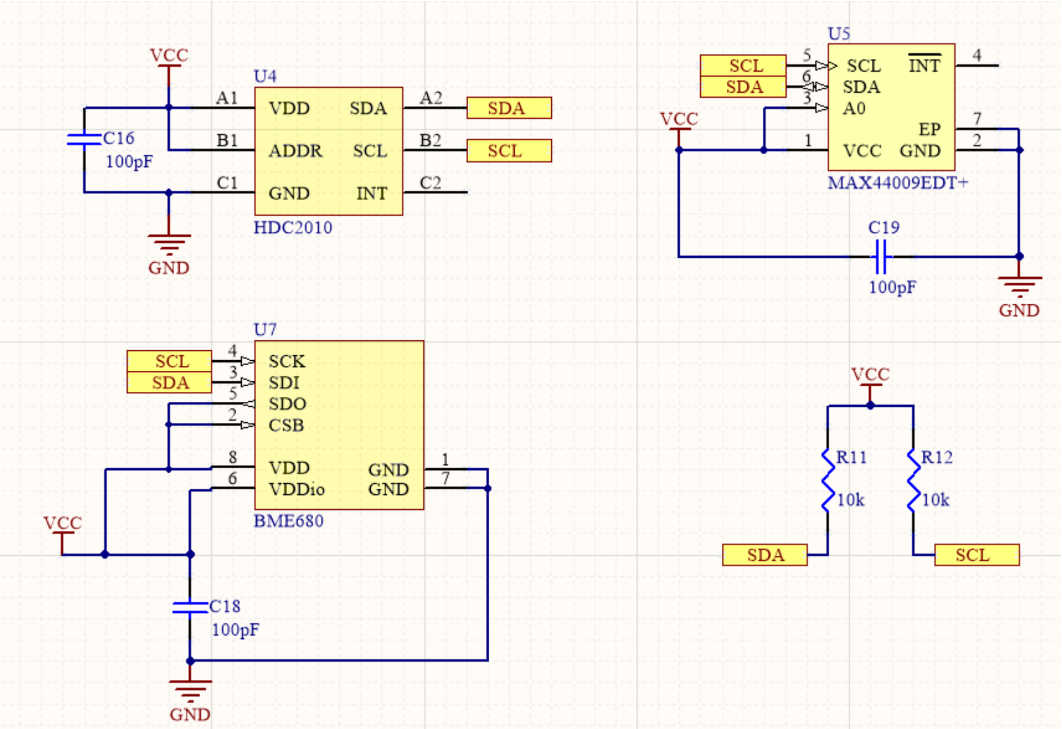


Figure 3.23: Supported sensors of the ATmega328p node

Figure 3.24 depicts the final PCB design of the sensornode, measuring a total of $49.5x30mm$. All physically tall and easily hand soldered components are placed on the bottom side in order to fascilitate for a smaller box for the nodes. The bottom side components are the battery holder, pin headers and the MAX44009 light sensor along with the light pipe. All Surface Mount Device (SMD) components, with the exception of the MAX44009, are placed on the top side of the PCB to allow for simple application of solder paste by using a stencil, simple placement of SMD components and to fascilitate for soldering by the use of a reflow oven. The antenna of the nRF24l01+ is placed as far away from traces and pads as possible to avoid interference and attenuation of the $2.4GHz$ radio signals. Additionally, the FR4 dielectric substrate is removed in the surroundings of the antenna. The unused areas of the PCB contain grounded planes in order to remove unwanted noise.

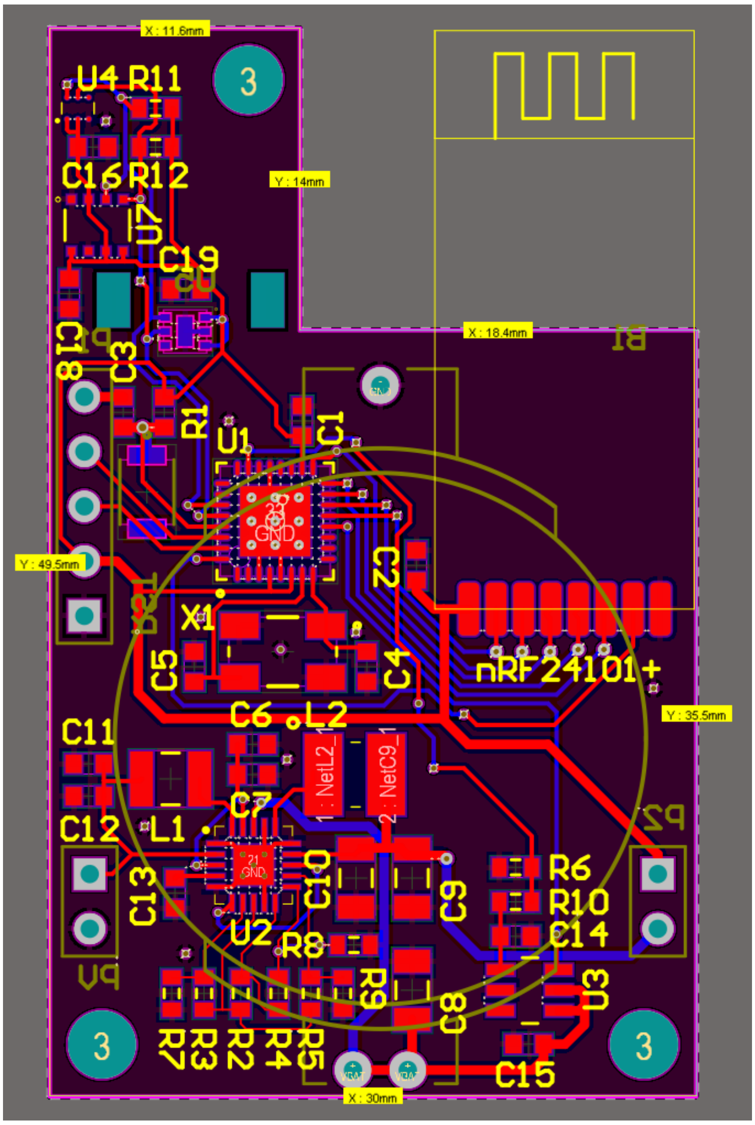


Figure 3.24: ATmega328p node PCB

3.4.1.3 Expected Energy Consumption

Table 3.11 presents the worst case current and power consumption of each component in the ATmega328p design.

Table 3.11: Expected worst case on-state energy consumption of ATmega328p node

Component	**Current**	**Power**
ATmega328p [48]	$3.58mA$	$11.8mW$
BME680 IAQ	$15mA$	$49.5mW$
BME680 Temperature	$350\mu A$	$1.2mW$
BME680 RH	$450\mu A$	$1.5mW$
BME680 Pressure	$849\mu A$	$2.8mW$
HDC2010 Temperature	$730\mu A$	$2.4mW$
HDC2010 RH	$890\mu A$	$2.9mW$
nRF24l01+PA+LNA High	$115mA$	$379.5mW$
Battery level	$210\mu A$	$693\mu W$
MAX44009	$650nA$	$2.1\mu W$

Table 3.12 presents the expected worst case sleep current and power consumption of each component in the design, as well as the total consumption of the entire node when using BME680 or HDC2010.

Table 3.12: Expected worst case sleep energy consumption of ATmega328p node

Component	**Current**	**Power**
ATmega328p watchdog timer enabled	$15\mu A$	$49.5\mu W$
BME680	$1\mu A$	$3.3\mu W$
HDC2010	$100nA$	$330nW$
nRF24l01+PA+LNA	$4.2\mu A$	$13.9\mu W$
Battery level	$100nA$	$330nW$
MAX44009	$650nA$	$2.1\mu W$
Total with HDC2010	$20.1\mu A$	$66.2\mu W$
Total with BME680	$21.0\mu A$	$69.1\mu W$

3.4.1.4 Enclosure

The enclosure for the sensor nodes are designed with the 3D modeling tool Autodesk Inventor 2017. It is composed of a backlid with standoffs and screw holes for mounting the PCB, and a shell that snaps onto the backlid. The front of the shell is made so that a $53x30mm$ panel can be snapped into place. The panel should be secured with glue, as there are small deviations in size of the panels. The shell has air vents on both sides near environmental sensors, a hole for the light pipe protruding from the PCB, and a notch on the bottom to enable detachment from the backlid. Multiple shells with different heights have been designed to accommodate for both button cell batteries and supercapacitors with different dimensions and capacities. The enclosure is designed to be produced on fused deposition modelling 3D printers without the need for supports. Figure 3.25 depicts an exploded view of the enclosure with panel and PCB. The external dimensions of the finished enclosure configured for a LIR2450 button cell battery is (H x W x D) $51.5mm \times 31.5mm \times 19mm$.

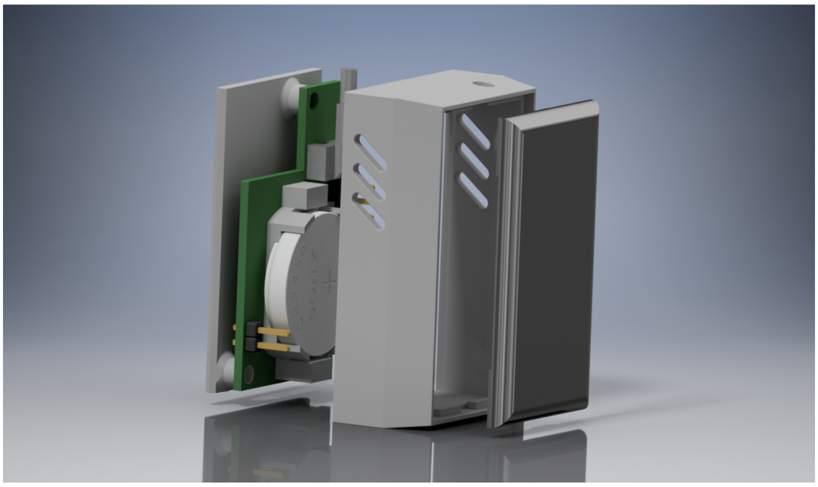


Figure 3.25: ATmega328p node enclosure

Figure 3.26 displays the enclosure with a bracket that supports a larger solar panel. The bracket snaps into place instead of the small solar panel. The bigger panel is glued onto the bracket.

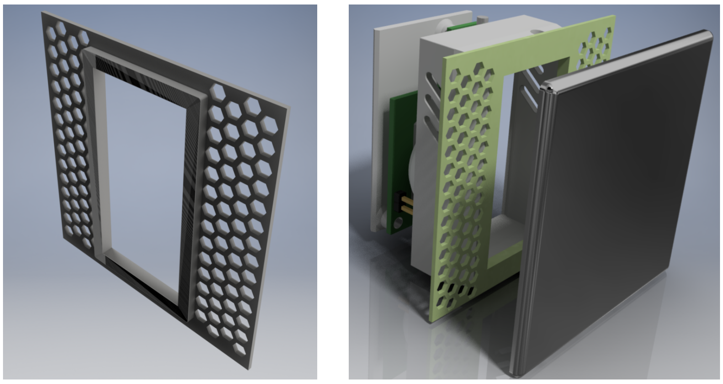


Figure 3.26: Bracket for larger solar panel

3.4.1.5 Finished Node

Figure 3.27 shows the assembled enclosure with the polycrystalline PV panel and the PCB inside. The enclosure is printed with polylactic acid with the Snapmaker with a 200μm layer height.

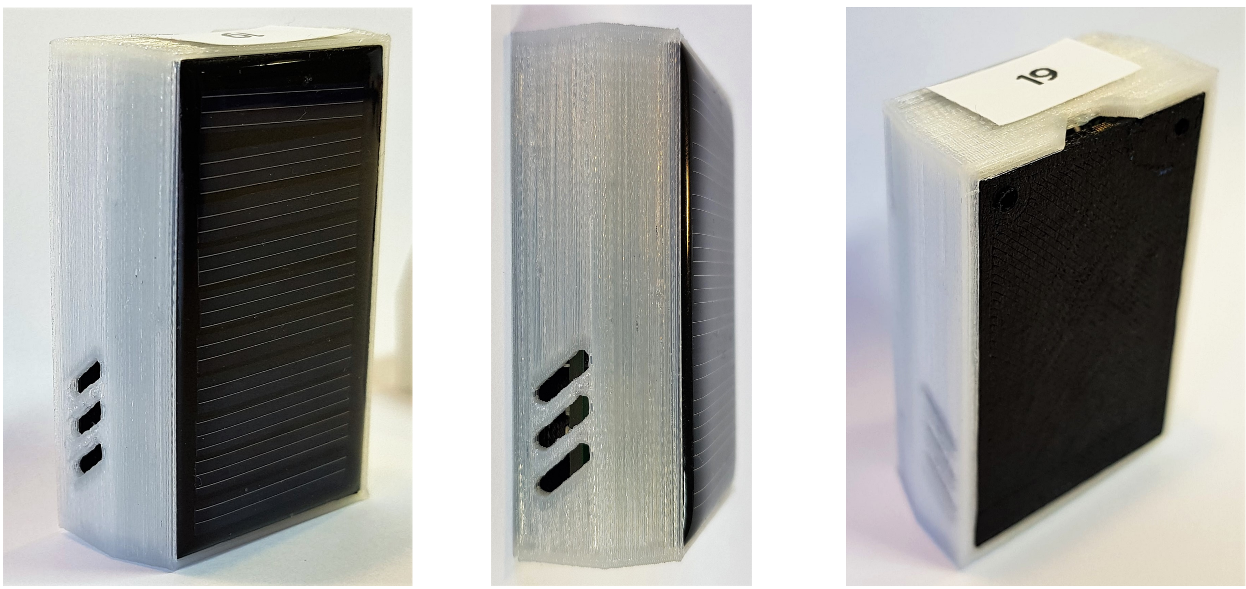


Figure 3.27: Printed enclosure

Figure 3.28 depicts the top side of the finished ATmega328p node. The top side is soldered by applying solder paste with a stencil, placing the components and heating the PCB up in a reflow oven. The entire process is described in appendix C.

The bottom side of the PCB is shown in figure 3.29, where the components are soldered by hand.

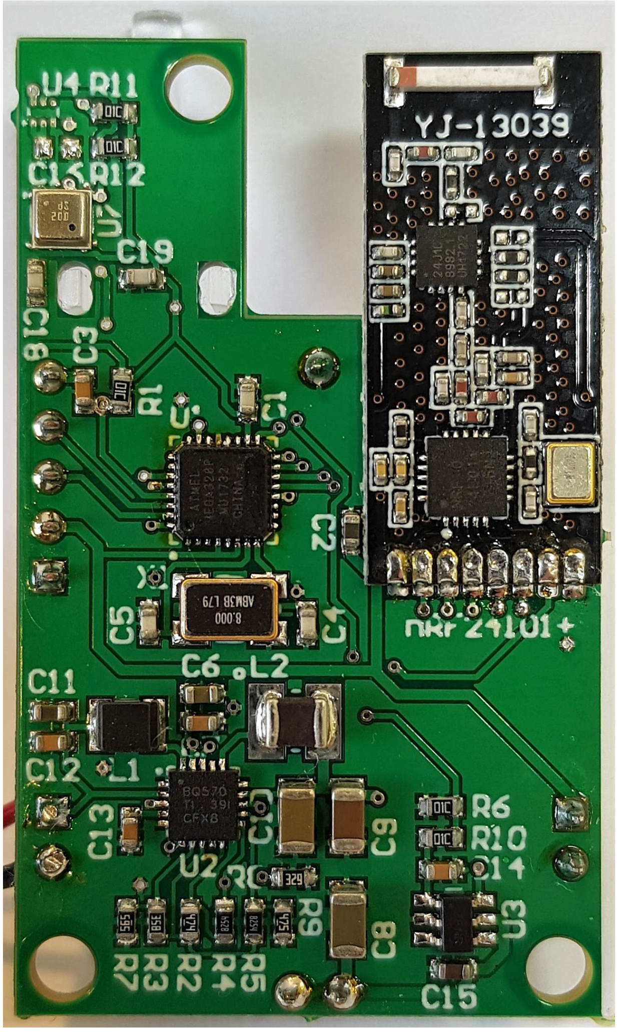


Figure 3.28: ATmega328p PCB top

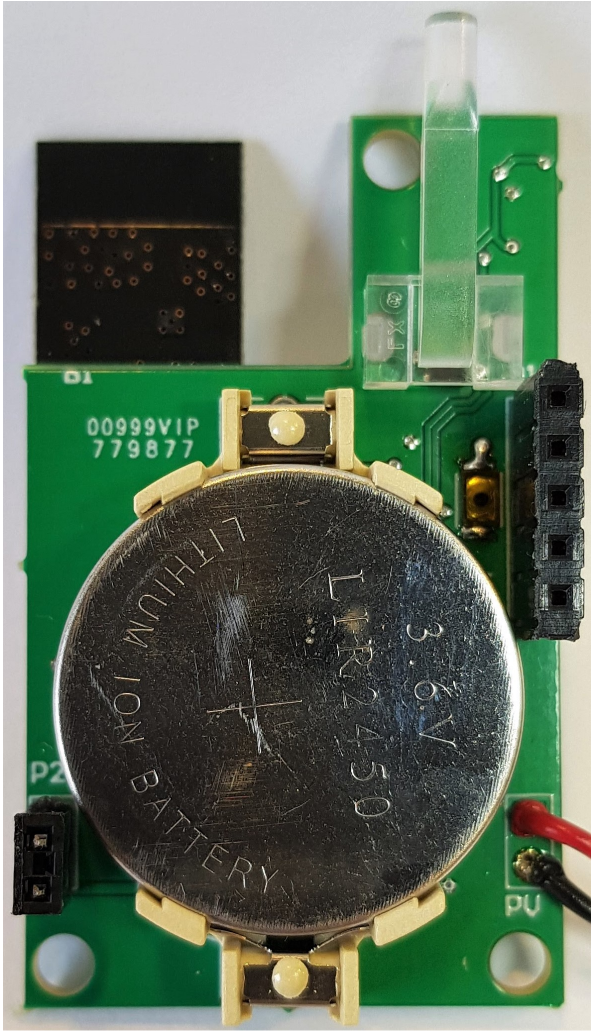


Figure 3.29: ATmega328p PCB bottom

3.4.2 ATmega328p Firmware

Two firmwares are created for the ATmega328p nodes; one for star topology and one for multi-hop topology. The two firmwares are for the most part identical regarding sensor readings, however differ in data communication methods.

3.4.2.1 Battery Level Sensing

In order to sense the battery voltage level, the sensing circuit consisting of a voltage divider and a switch, described in section 3.2.4, is turned on. The ON-pin of the TPS22860 switch is connected to the the digital pin 8 on the ATmega328p, which is pulled *HIGH*. The voltage after the voltage divider is sampled ten times, then averaged, in order to increase the precision of the reading. After sampling is completed, the ON-pin of the switch is returned to *LOW* state, cutting the voltage to the voltage divider, effectively saving power. The built-in ADC in the ATmega328p has a resolution of 10 bit, and uses VCC as a reference, allowing for voltages to be read in the range of $0V$ to $3.3V$, resulting in each step having a value of $\sim 0.003V$. Therefore, the voltage measured by the ADC can be calculated by equation 3.17.

$$V_{BAT} = \frac{SAMPLE_{AVG}}{1024} \cdot 3.3V \cdot \left(\frac{10k\Omega + 10k\Omega}{10k\Omega}\right) \tag{3.17}$$

Where:

- $SAMPLE_{AVG}$ is the average of the 10 samples
- 1024 is the resolution of the ADC
- 3.3V is the voltage reference of the ADC
- $\left(\frac{10k\Omega + 10k\Omega}{10k\Omega}\right)$ is used to multiply the divided voltage in the voltage divider

3.4.2.2 I^{2}C Scanner

The node is designed for the ability to handle various types of I^{2}C sensors, namely the BME680, HDC2010 and MAX44009. It can automatically detect which sensors are connected by utilizing an I^{2}C scanner, the functionality for which is shown in figure 3.30.

The I^{2}C scanner checks every I^{2}C address in the range from 0 to 127 at startup in order to see if any devices respond. If a response is received, it compares the responding address with a list of sensor addresses. If the address is recognized, the sensor is enabled in the code, meaning it will be utilized by the node. Devices not detected in the I^{2}C scanner will not be utilized by the node. This is done by enveloping the corresponding code in an *if* statement, as shown below, so that unused sensor code is not run.

```
if (i2c_device[sensorname_i2c][1]) {
    sensor code...
}
```

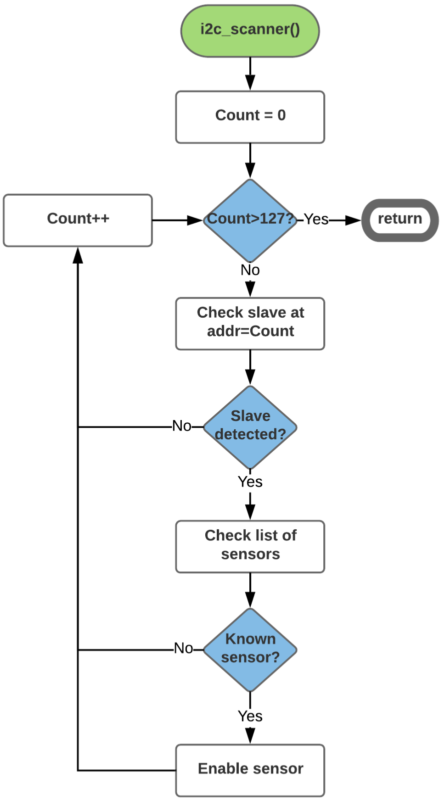


Figure 3.30: Flowchart of *i2c_ scanner()*

3.4.2.3 BME680

Interfacing the BME680 on the ATmega328p is done with the help of the *ClosedCube_-BME680.h* library, available on github [49]. The code for BME680 can be found in appendix A.1.1, and an overview of its register addresses in section 3.2.1. Two main functions are called to configure the BME680. *bme680.setOversampling()* decides how many samples will be taken for humidity, temperature and pressure. If set to zero, the measurement will be skipped. If set to a high number, more samples are taken for increased accuracy, but also with increased response time. Gas measurement is enabled separately with the function *bme680.setGasOn()* which is configured with user defined temperature and duration for the gas heater. *bme680.setIIRFilter()* is used to suppress anomalies and give a more stable output. Enabled measurements will consecutively be performed one single time when *bme680.setForcedMode()* is called. Data is then stored in corresponding registers, and the BME680 returns to sleep.

3.4.2.4 HDC2010

Four different functions are created to interface with the HDC2010. *hdc2010_init()* is used at startup, writing 0x00 to the configuration register 0x0E, which keeps the heater off and enables data measurements on trigger only. *hdc2010_ask_data()* writes 0xA1 to the measurement configuration register 0x0F, which starts measurements for both temperature and humidity values, both at a 9 bit resolution, for the fastest conversion time. 11 bit and 14 bit can also be set, however this will increase the conversion time, resulting in higher energy consumption. *hdc2010_read()* reads four consecutive bytes from register 0x00 through 0x03. 0x00 and 0x01 contain the Most Significant Bit (MSB) and Least Significant Bit (LSB) of the temperature measurement, while 0x02 and 0x03 contain the MSB and LSB of the humidity measurement. The full code can be seen in appendix A.1.2, and register addresses of the HDC2010 in section 3.2.2.

3.4.2.5 MAX44009

The default configurations of MAX44009 are originally configured to continuous read mode and 100ms integration time, which are the preferred settings. Therefore, no additional configuration is needed. *max_read()* is therefore the only function used, which reads two bytes from register 0x03, containing the data required in order to calculate the LUX value. The code for MAX44009 can be found in appendix A.1.3, and its register addresses in section 3.2.3.

3.4.2.6 get_sensor_data()

The *get_sensor_data* function, which is described in the flowchart in figure 3.31, is used to obtain sensor data and sense battery level. The sensors detected from the I^{2}C scan described in section 3.4.2.2 determines which sensors will be enabled in the firmware, and therefore read with this function. When asking for sensor data from each of the I^{2}C sensors, a measurement and conversion time period is required before the data are available to be read by the MCU. Therefore, the sensor which has the longest measurement period is asked first, and read last to allow other tasks to be performed while the sensor is generating data. The BME680 takes approximately 12 milliseconds to measure and convert temperature, pressure and humidity, which is the slowest, and is therefore asked for data first. A timer is simultaneously started, which is used to make sure the new value is completed before reading the data register. On every tenth call of *get_sensor_data()*, gas measurement is also performed. This will increase the total measurement time of the BME680 by the duration of the heating period, which is $200ms$, and it will therefore take longer for the BME680 to measure and convert data. Battery level sensing is done in between asking the sensors for data and reading it, utilizing time where the MCU would otherwise have been idle and waiting for data.

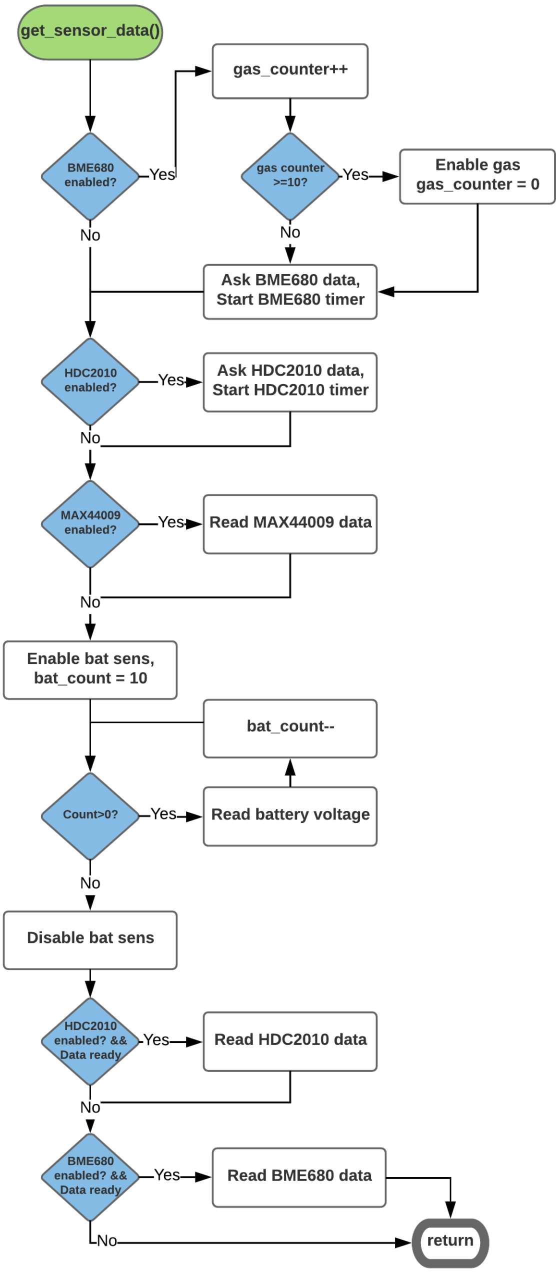


Figure 3.31: Flowchart of *get_sensor_data()*

3.4.2.7 Sleep

Sleep mode on the ATmega328p is accomplished by using the watchdog timer, which runs on a separate, low power $128kHZ$ oscillator. This allows the MCU to power down, and let the watchdog timer wake it back up. The watchdog timer is however limited to a maximum interval of 8s for a timed wake interrupt. This means that for sleep intervals longer than 8s, the clock must be restarted in order for the MCU to return to sleep.

To interface the watchdog timer, the library *LowPower.h* available on github is used [50]. It supports continuous sleep times up to 8s while also turning off brown-out detection and ADC, which further reduces power consumption in sleep. Brown-out detection is a feature implemented in many MCUs which senses a sudden voltage drop in the power supply. This lets the MCU reset so that no unforeseen lockups can occur because of a voltage drop [51]. The brownout detection is however not necessary for the ATmega328p node, as the BQ25570 is programmed to disable its buck converter whenever the battery or supercapacitor is discharged to levels where it is unable to supply sufficiently high voltage. The watchdog timer is constrained by a set of available timers. For values from one second and above, these are $8s$, $4s$, $2s$ and $1s$, however the range goes all the way down to $15ms$. A function is designed which allows any chosen sleep value to be represented by these whole second values as shown in code 3.1

Code 3.1: Sleep function

```
void rtc_sleep(uint16_t sleeptime) {
  uint16_t remainder = sleeptime;
  uint8_t cycles;
  while (remainder != 0) {
    if (remainder >= 8) {
      cycles = remainder / 8;
      remainder = remainder % 8;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF);
      }
    }
    else if (remainder >= 4) {
      cycles = remainder / 4;
      remainder = remainder % 4;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF);
      }
    }
    else if (remainder >= 2) {
      cycles = remainder / 2;
      remainder = remainder % 2;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
      }
    }
    else if (remainder >= 1) {
      cycles = remainder / 1;
      remainder = remainder % 1;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_1S, ADC_OFF, BOD_OFF);
      }
    }
    else {
      remainder = 0;
    }
  }
}
```

3.4.2.8 nRF24l01+

Interfacing the nRF24l01+ is done using the *RF24.h* library, available on github [52]. This library contains functions that simplifies nRF24l01+ operations. Code 3.2 shows the parameters which are set for the ATmega328p nodes.

Code 3.2: nRF24l01+ Parameters

```
void radio_init() {
  radio.begin();
  radio.setCRCLength(RF24_CRC_16);
  radio.setPALevel(RF24_PA_LOW);
  radio.setDataRate(RF24_1MBPS);
  radio.setChannel(0x76);
  radio.enableDynamicPayloads();
  radio.setRetries(0, 0);
  radio.setAutoAck(false);
  radio.openWritingPipe(pipe);
  radio.openReadingPipe(1, pipe);
  radio.powerDown();
}
```

The parameters in code 3.2 control how the radio operates. *setChannel()*, *setDataRate()* and *setPayloadSize()* should be the same for all transceivers in order for them to communicate with each other. Both *setPALevel()*, which determines how much power the radio is allowed to use, and *setRetries()*, which determines how many retries the transceiver will make after failed transmissions, can be modified to meet the desired performance. *openWritingPipe()* decides to which address data will be sent. This can only be one address at a time, and must be changed in order to transmit to a new node. There are, however, 8 pipes in which the transceiver can store packages from, meaning that it can receive packets on 8 different channels, although not at the same time as this would cause collisions. Finally, *powerDown()* is set to turn the radio off while not used.

3.4.2.9 Star topology

The star topology loops the three functions in code 3.3. *get_sensor_data()* and *rtc_sleep()* are explained in section 3.4.2.6 and 3.4.2.7 respectively.

Code 3.3: Star

```
void loop() {
  get_sensor_data();
  send_data();
```

```
    rtc_sleep(SLEEP_TIME);
  }
```

Transmitting data is done with the following four functions, where *send_data()* initiate the transmission:

- *carrier_sense()*
- *send_data()*
- *wait_for_ack*
- *adjust_pa()*

send_data(), shown in figure 3.32 is called whenever data should be transmitted. The function powers up the radio and calls *carrier_sense()* to see if the channel is idle before attempting to transmit data to reduce collision probability. After transmitting, *wait_for_ack* is called, which waits for up to three milliseconds for an ACK from the gateway. If an ACK is not received, it will retransmit the packet. A maximum of four retransmissions are attempted before the data is discarded, and the node goes back to sleep. If a transmission fails to be sent for more than four consecutive transmission intervals, however, the function *adjust_pa()* is called, which automatically adjusts the power level until it is able to transmit successfully to the gateway. The power level is set to the minimum power value, and is incremented for each unsuccessful transmission until a packet is successfully sent. This is because the nRF24l01+ can be sensitive to high power signals, thus a lower power level may be beneficial. Finally the radio is powered off before exiting the function. A description of the gateway can be seen in section 3.5.

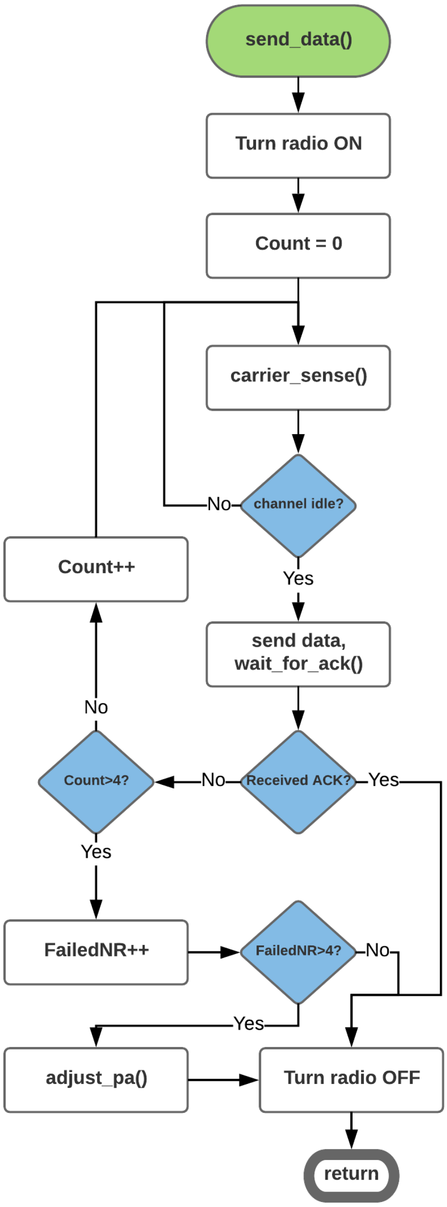


Figure 3.32: Flowchart of *send_data()*

3.4.2.10 Multi-hop topology

The multi-hop topology main loop loops the code shown in 3.4. The nodes are generally in sleep state, but every 2 seconds, either *wake_ping()* or *get_sensor_data()* and *wait_to_send()* is called. The user defined number *SEND_INTERVAL* determines the interval of the nodes own data transmission, which occurs after a set number of *wake_ping()* intervals. When this number is reached, the node will call *get_sensor_data()* instead of *wake_ping()*. *get_sensor_data()* reads sensor data before it enters the *wait_to_send()* function. The full multi-hop code can be found in appendix A.2.

Code 3.4: Multi-hop

```
void loop() {
  if (send_counter < SEND_INTERVAL) {
    wake_ping();
  }
  else {
    get_sensor_data();
    wait_to_send();
    send_counter = 0;
  }
  send_counter++;
  radio.powerDown();
  LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
  radio.powerUp();
}
```

The multi-hop topology code is based on five main functions;

- *wake_ping()*
- *wait_for_response()*
- *send_ack()*
- *wait_to_send()*
- *wait_for_ack*

The *wake_ping()* function broadcasts a packet containing its own ID and number of hops to the gateway in a struct.
After a broadcast message is transmitted, the node enters the *wait_for_response()* function as shown in figure 3.33, which causes it to wait for up to 10ms for anyone to transmit a data packet to it. If a data packet is received with its own ID attached, the data message will be stored in the *data_msg* struct and the hops variable will be incremented by one, and an ACK is sent back to the sender. The node will then call the function *wait_to_send()* in order to relay that message toward the gateway node.

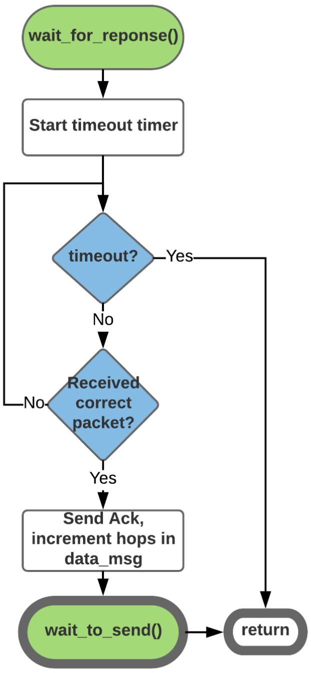


Figure 3.33: Flowchart of *wait_for_response()*

wait_to_send() function shown in figure 3.34 is called when the node is either sending its own data, or relaying received data. If the node is adjacent to the gateway, *hops_to_gateway* is equal to one, and it does not have to wait for a wake ping to send the data as the gateway node is always on and ready to receive data. It will however perform carrier sensing in order to determine if the channel is idle before sending the packet, and wait for up to $3ms$ for an ACK after transmission. If no ACK is received after six consecutive transmission attempts, its *hops_to_gateway* will be incremented by one, and it will wait for a wake ping instead of attempting to send directly to the gateway.

Nodes with *hops_to_gateway* larger than one will wait for wake ping messages with *hops_to_gateway* lower than itself. Since a wake ping should be received within a maximum of $2s$, a timeout is implemented at $2.2s$ if no wake pings are received. Each time *wait_to_send()* times out, the nodes *my_hops_to_gateway* variable is incremented, increasing the probability of finding an adjacent node to relay its data, and decreasing the probability of relaying data from other nodes.

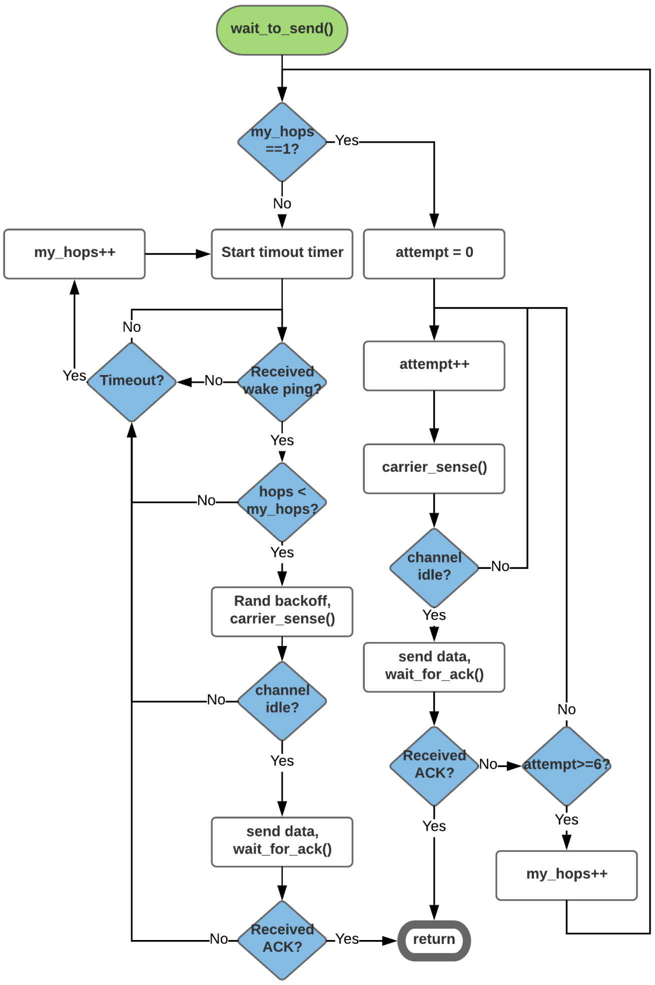


Figure 3.34: Flowchart of *wait_to_send()*

Whenever a wake ping is received with a hops to gateway value lower than its own, the node will set its *my_hops_to_gateway* equal to the received hop count plus one, and store the ID of the wake ping source in *next_hop*. Before transmitting its data to the relaying node, a random backoff is performed prior to executing a single $128\mu s$ carrier sense in order to reduce the probability of collision at the receiver. If the channel is idle after the backoff, the data packet is transmitted, and the node calls the *wait_for_ack()* function. If an ACK is received with the correct ID, the node breaks the loop and exits the function. However, if the channel is busy or if no ACK is received, the node will keep waiting for a wake ping, to be able to send its data.

3.5 ATmega328p Gateway

The gateway for the ATmega328p consists of an ESP32 MCU, which is developed by Espressif [53]. The ESP32 offers a dual core processor and built-in Wi-Fi, and is supported by the Arduino environment. An nRF24l01+PA+LNA is connected to the ESP32 in order to gather data from the nodes. Transmissions from the node are received by the nRF24l01+PA+LNA, and forwarded via Wi-Fi to an IoT server in order to log and display the sensor data. The gateway is powered by USB in order to have sufficient power for constant operation as the current consumption is in the order of hundreds of mA. Figure 3.35 depicts the prototype of the gateway.

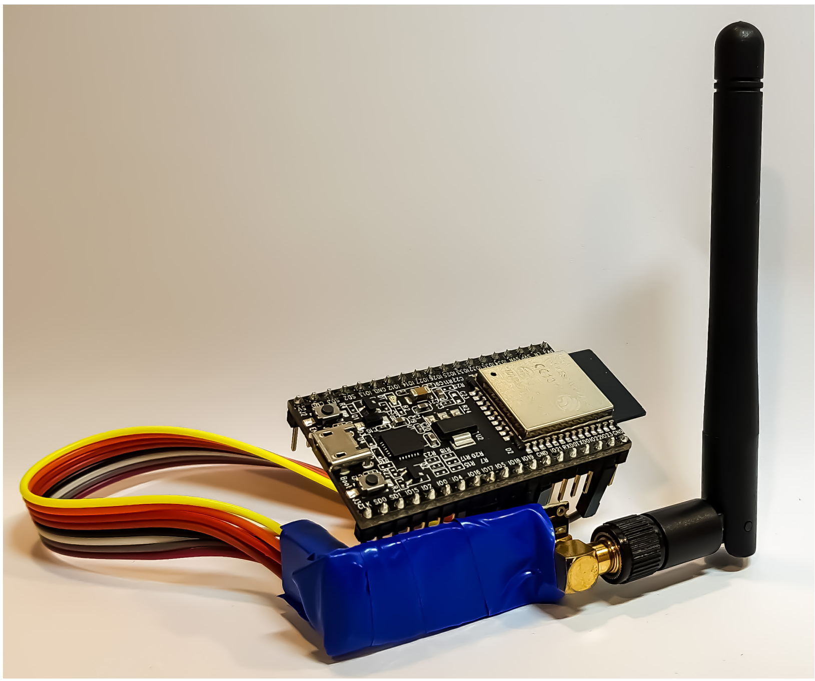


Figure 3.35: Gateway prototype for ATmega328p nodes

3.5.1 ATmega328p Gateway firmware

nRF24l01+

The nRF24l01+ is initialized in the same way on the gateway node as it is on any other node, which is shown in code 3.2, except the power level is set to MAX. The gateway does however responds to incoming messages with the same power level as the transmitting node.

In the gateway main loop, the radio is always waiting for packets, as seen fig 3.36. Only packets with a correct packet size will be stored. If a duplicate packet is received, which is identified by receiving the same packet number for the same node twice, it will be discarded, however an ACK is sent to let the node know that the packet is received previously. An ACK is also sent upon receival of new data packets. The *send_ack()* function checks the received packet power level, and sets its own power level to the same value before responding with the ACK, effectively reducing interference. Finally, the packet number is stored in an array in order to allow for duplicate checks for new data packets before publishing to MQTT. Full code for the ATmega328p Gateway can be found in appendix A.3.

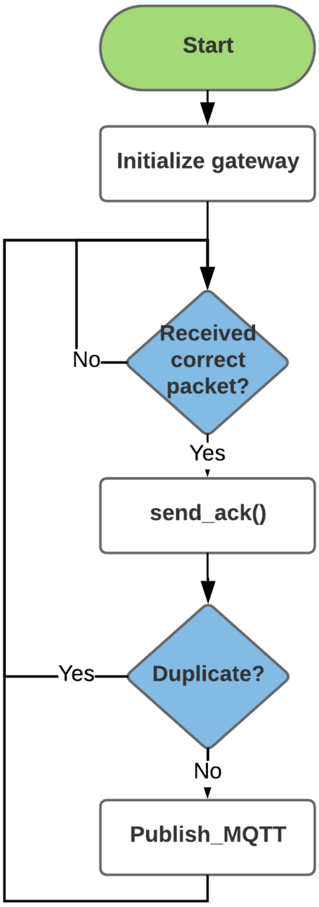


Figure 3.36: Gateway functionality flowchart for ATmega328p nodes

MQTT

Before attempting to publish to the MQTT broker, the Wi-Fi and MQTT connection is first verified. If Wi-Fi is not connected, *setup_wifi()* is called, and if MQTT client is not connected, *reconnect()* is called. The data is then published to the MQTT broker. The full gateway code is found in appendix A.3. MQTT functions are from *PubSubClient* library available on GitHub [54].

3.6 nRF52840 Node

3.6.1 nRF52840 Hardware

The nRF52840 is a System on Chip (SoC) developed by Nordic Semiconductor, which is developed for ULP wireless performance with Bluetooth 5 support, and is built around a 32-bit Cortex-M4F processor [55].

At the time of the project, the IC for the nRF52840 is still in production and is therefore unattainable, however the nRF52840-PDK is available. As a result, the PDK is used for development and evaluation. A PCB is however designed and ready to be ordered. The PCB contains the BMD-340 module developed by Rigado, which is based on the nRF52840 SoC [56]. It is selected because of its on-board antenna, its small footprint and the possibility of programming directly via a USB connection, UART or Over-the-Air (OTA).

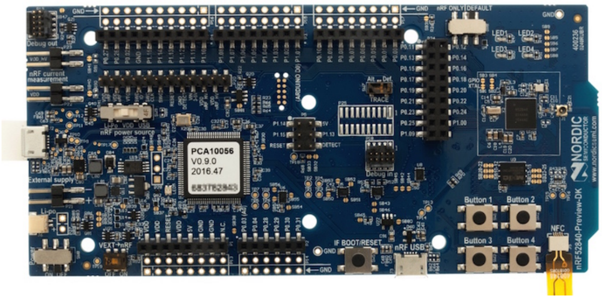


Figure 3.37: nRF52840-PDK [3]

Figure 3.38: BMD-340

3.6.1.1 Expected Energy Consumption

Table 3.13 presents the worst case current and power consumption of each component in the design. Since the operation with 1.9V supply proved superior to 3.3V, 1.9V is used for the nodes, and tables 3.13 and 3.14 are based on current and power consumption with 1.9V

Table 3.13: Expected worst case on-state energy consumption of nRF52840 node

Component	**Current**	**Power**
BMD-340 CPU at 64MHz	6.5mA	12.4mW
BMD-340 +8dBm TX	27.6mA	52.4mW
BMD-340 +8dBm RX	11.7mA	22.2mW
BME680 IAQ	15mA	28.5mW
BME680 Temperature	350μA	665μW
BME680 RH	450μA	855μW
BME680 Pressure	849μA	1.6mW
HDC2010 Temperature	730μA	1.4mW
HDC2010 RH	890μA	1.7mW
Battery level	57.4μA	109.1μW
MAX44009	650nA	1.2μW

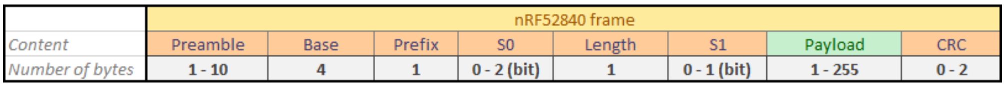
Table 3.14 presents the expected worst case sleep current and power of each component in the design

Table 3.14: Expected sleep energy consumption of nRF52840 node

Component	**Current**	**Power**
BMD-340[i]	3.2μA	720nW
BME680	1μA	6.1μW
HDC2010	100nA	180nW
Battery level	100nA	180nW
MAX44009	650nA	1.2μW
Total with HDC2010	4.1μA	7.7μW
Total with BME680	5.0μA	9.4μW

[i] With RAM retention of $64 \cdot 4KB$ blocks

Figure 3.39 depicts the on-air message frame for the built in transceiver.



	nRF52840 frame							
Content	Preamble	Base	Prefix	S0	Length	S1	Payload	CRC
Number of bytes	**1 - 10**	**4**	**1**	**0 - 2 (bit)**	**1**	**0 - 1 (bit)**	**1 - 255**	**0 - 2**

Figure 3.39: nRF52840 data frame

Figure 3.40 displays the enclosure for the nRF52 PDK evaluation node, which has been designed to protect the PCB during handling and testing.

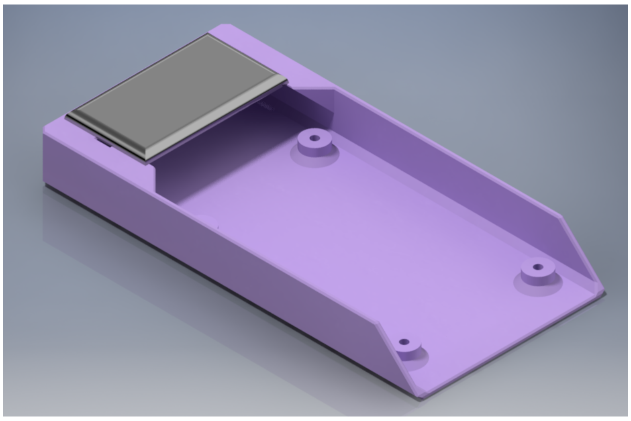


Figure 3.40: Protective enclosure for nRF52840 PDK

3.6.1.2 Schematic and PCB Design

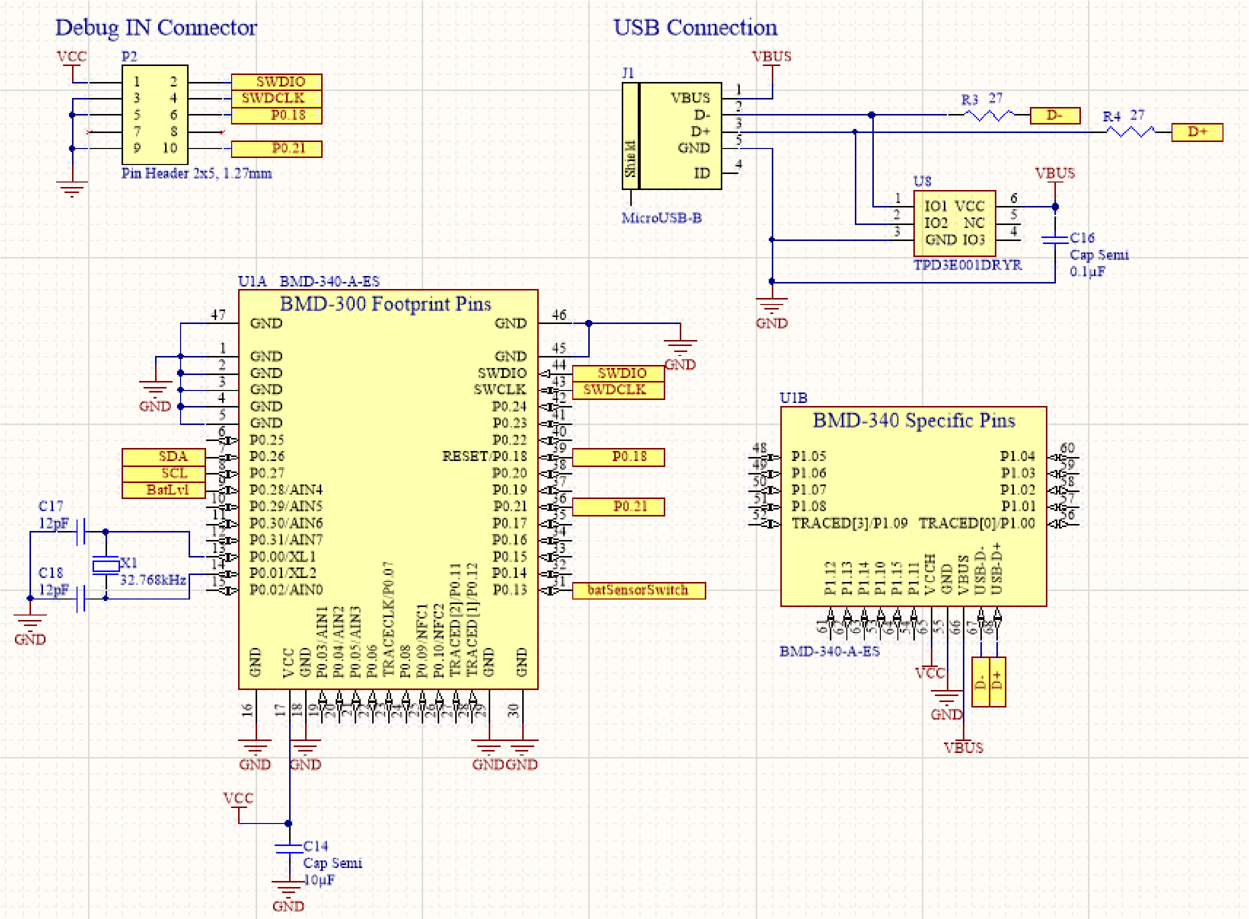


Figure 3.41: Schematic of the nRF52840 node

Figure 3.41 shows part of the circuit for the nRF52840 sensor node. The BMD-340 (U1) has an internal high frequency clock. A low frequency Real-Time Clock (RTC) can be synthesized from the high frequency clock, however an oscillator should be connected externally for optimal stability. Therefore, a $32.768kHz$ (X1) clock is implemented with suitable decoupling capacitors. A 2-row, 10-pin header with $1.27mm$ pitch is included for interfacing with the Segger J-Link debugger. Additionally, the BMD-340 is programmed with the Rigado RigDFU bootloader at the factory, allowing for programming via UART, or directly via USB. Therefore, a MicroUSB connector is included, along with the electrostatic discharge protection IC and passive components according to the recommendations in the BMD-340 datasheet. All schematics for the nRF52840 node, as well as the PCB, can be found in appendix D.2.

The energy harvesting circuit and sensor circuits are identical to the ones in the ATmega328p node design seen in section 3.4.1.2 with different resistors for programming the BQ25570, however as the nRF52840 node already has a USB connection, the possibility of charging the battery via USB is added, the circuit for which is depicted in figure 3.42. As the BQ25570 does not allow for simultaneous charging from an external source, a transistor (Q1) is added as a security measure. VBUS is connected to the base of the transistor, so whenever a USB connection is made, voltages of up to $5.5V$ is applied, turning on the transistor. The battery is connected to the collector, and when the transistor is on, the BQ25570 $\overline{EN}$ pin (BQ_EN) is pulled high, disabling the BQ25570 IC entirely as soon as the USB is connected to a power source. The VBAT pin of the BQ25570 is connected through a MOSFET internally, so no additional security is required.

The low-cost MCP73831T battery charger (U9) is implemented [57]. Its maximum output current is limited to $83.3mA$ by using a $12k\Omega$ resistor between its PROG pin and ground, as calculated by equation 3.18 given by the datasheet.

$$I_{REG} = \frac{1000V}{R_{PROG}} = \frac{1000V}{12k\Omega} = 83.3mA \tag{3.18}$$

Additionally, the MCP73831T implements a status pin, which is HIGH while charging, and LOW when finished. Therefore, a red LED (LED1) is added with a current limiting resistor (R14). Red LEDs commonly have a forward voltage of $2V$ and a maximum forward current of $20mA$. Maximum brightness is however not needed, and so the current is set to $12mA$. As such, the resistance is calculated by $R14 = (5.5V - 2V)/12mA = 291\Omega \approx 300\Omega$. As the power dissipated by the resistor will be $P_{R14} = (5.5 - 2V) \cdot 12mA = 147mW$, a 1206 resistor is used, with a power rating of $250mW$.

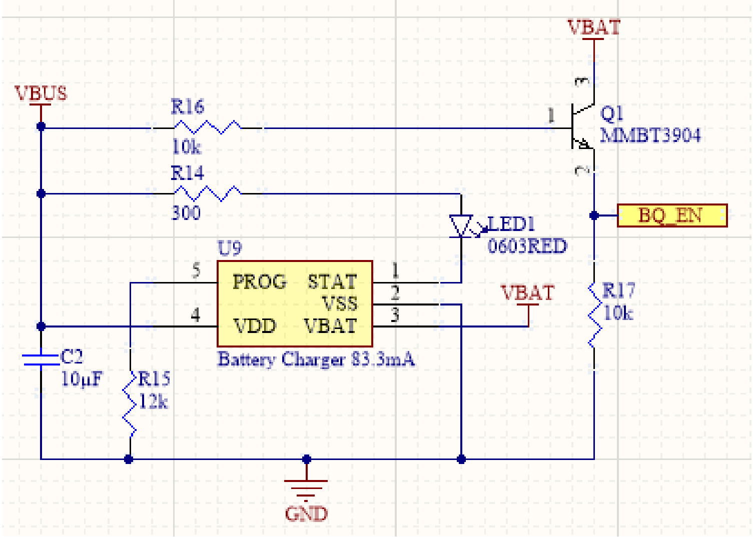


Figure 3.42: USB battery charger schematic

Figure 3.43 depicts the final PCB design of the nRF52840 sensor node. The design is similar to the that of the ATmega328p node, as described in 3.4.1.2. All SMD components, with the exception of MAX44009, are placed on the top side of the PCB. The antenna of the BMD-340 is placed according to the datasheet, with no copper on either side of the PCB for its entire width. The dimensions and mounting holes are the same as the ATmega328p node so the same enclosure can be used. Additionally, improvements have been made on the PCB surrounding the HDC2010. The ground plane is removed on both sides of the PCB, and notches are created on both sides in order to maximize the precision of the sensor.

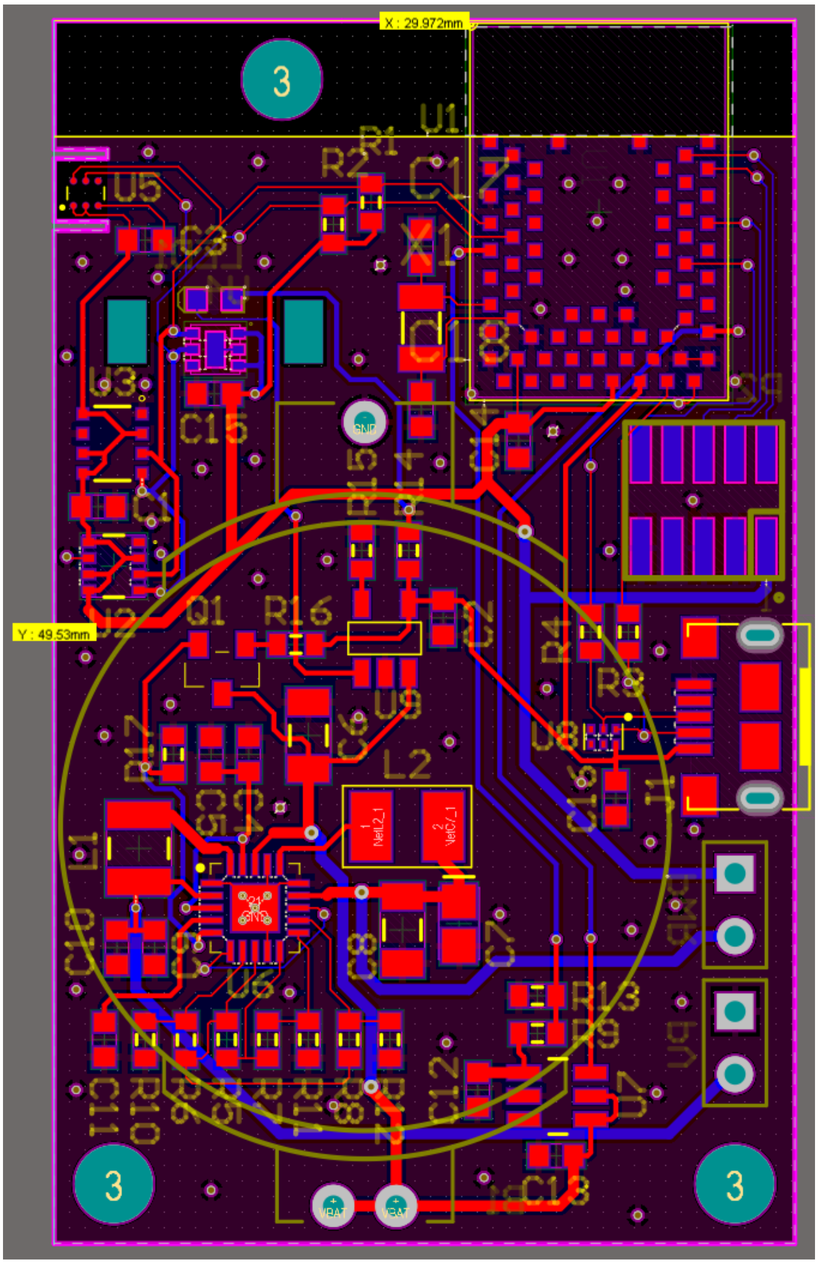


Figure 3.43: nRF52840 node PCB

As the nRF52840 IC is not released during the project, the PDK is used for testing. A simple breakout board, shown in figure 3.44, is designed based on the ATmega328p node circuit with the same pin layout as the PDK, allowing for simple connection. The breakout board contains the EH circuit as well as all the sensors, and the full design can be found in appendix D.3

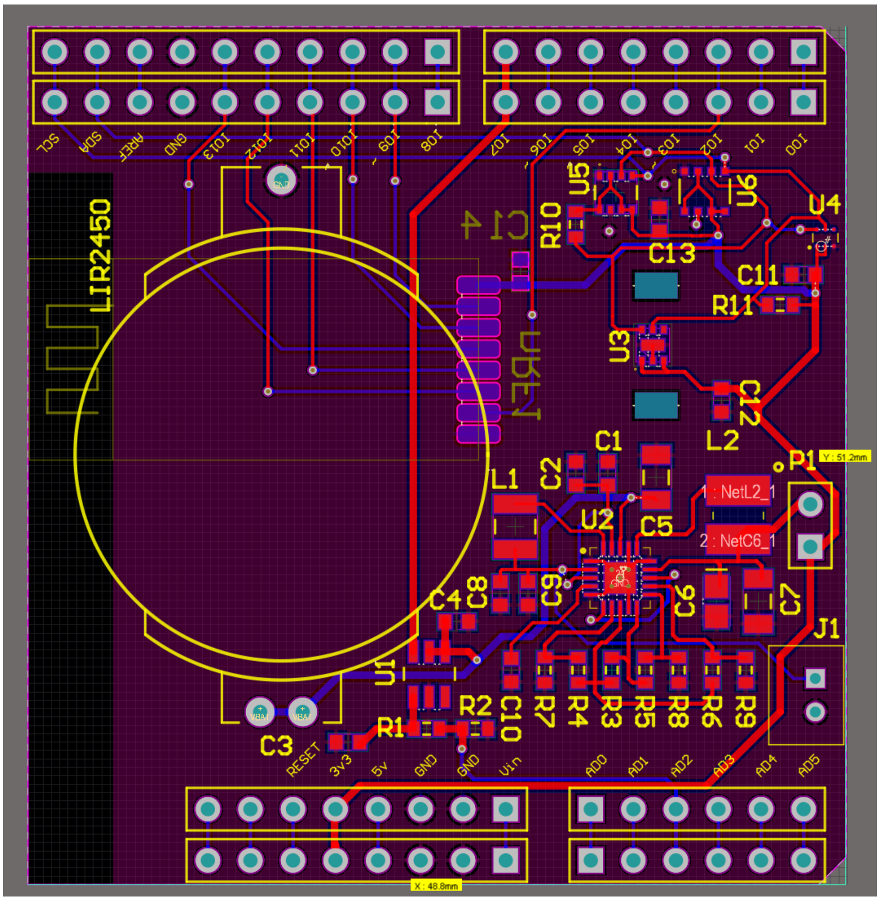


Figure 3.44: nRF52840 PDK breakout PCB

Figure 3.45 depicts the finished shield installed on the nRF52 PDK.

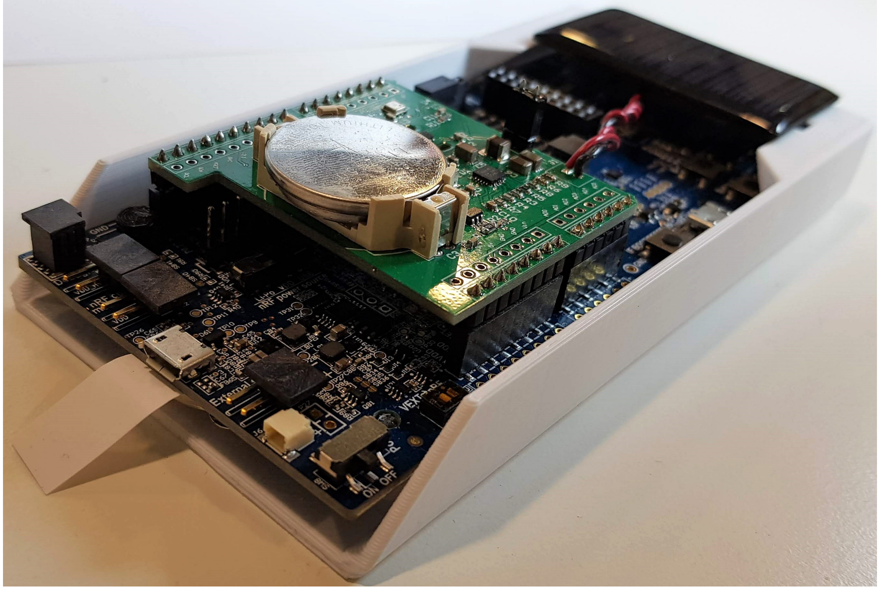


Figure 3.45: nRF52840 PDK in the 3D printed enclosure with breakout board installed

3.6.2 nRF52840 Firmware

Two firmwares are created for the nRF52840 nodes; one for star topology and one for multi-hop topology. The two firmwares are for the most part identical regarding sensor readings, however differ in data communication methods.

3.6.2.1 System Clocks and Interrupt Timers

When not using the SoftDevice provided by Nordic Semiconductor, which is a precompiled and linked binary software implementing a wireless protocol, the low frequency clock (LFCLK) and high frequency clock (HFCLK) need to be initialized. The HFCLK is used for fast processing and radio operations when the node is active, while the LFCLK runs continuously for application timer interrupts. The LFCLK runs on an external, low power 32.768kHz oscillator. Whenever there are no pending processes, the nRF52840 calls the *power_manage()* function, shown in code 3.5, which turns off the HFCLK and clears any pending interrupts which may keep the nRF52840 awake.

Code 3.5: Power manage

```
static void power_manage(void) {
  NRF_CLOCK->TASKS_HFCLKSTOP = 1;
  NVIC_ClearPendingIRQ(SAADC_IRQn);
  __set_FPSCR(__get_FPSCR() & ~(0x0000009F));
  (void)__get_FPSCR();
  NVIC_ClearPendingIRQ(FPU_IRQn);
  __SEV();
  __WFE();
  __WFE();
}
```

The *__SEV()* function detects if there are any pending interrupts as to make sure that the node is not immediately woken up. Following the *__SEV()* function are two *__WFE()*, the first of which responds to the *__SEV()* call, which may then wake the node up, and the second returns the node to low power mode.

The application timers are created on startup with the following structure:
app_timer_create(ID, MODE, TIMEOUT_HANDLER);
The *ID* is defined for each application timer in order to differentiate between them. *MODE* can be set to either repeat or to only run once. *TIMEOUT_HANDLER* specifies the desired functionality to run at timeout. *app_timer_start(ID, INTERVAL, POINTER);* is used to start the timer. *ID* selects the associated timer, *INTERVAL* sets the number of ticks before the timer expires and *POINTER* is a general purpose pointer that points to the *TIMEOUT_HANDLER*. Whenever the application timer interrupt is called, it will break the *power_manage()* function and execute the *TIMEOUT_HANDLER*.

3.6.2.2 Battery Level Sensing

The ADC of the nRF52840 is configured with the internal $0.6V$ voltage reference and an internal gain of 1. Battery level sensing on the nRF52840 is performed by using the same battery level sensing circuit as the ATmega328p as seen in 3.4.2, however with resistor values suitable for the $0.6V$ reference. The batSensorSwitch signal is connected to pin P1.08, and is pulled high to enable battery sensing. The ADC voltage sensing is, however, performed in a different manner. The ADC has to be initialized before every use, as it needs to be uninitialized for the MCU to enter sleep. When initialized, the analog read pin is set to AIN4 which is connected to the BatLvl pin from the voltage divider. *nrf_ drv_ saadc_ sample()* is used to start a battery measurement. When the sample is completed, a callback function is called, which retrieves and calculates the correct voltage from the ADC sample.

Equation 3.19 presents the calculation used in order to determine the battery voltage.

$$V_{BAT} = V_{IN} \cdot \left(\frac{REFERENCE}{GAIN \cdot 2^{RESOLUTION}} \right) \cdot \left(\frac{10k\Omega + 82k\Omega}{10k\Omega} \right) \tag{3.19}$$

Where:

- V_{BAT} is the measured voltage
- V_{IN} is the measured voltage represented in a 10b resolution
- $GAIN$ is the selected gain setting, here 1
- $REFERENCE$ is the selected reference voltage, here $0.6V$
- $RESOLUTION$ is the resolution of the ADC, here 10b
- $\left(\frac{10k\Omega + 82k\Omega}{10k\Omega} \right)$ is the multiplier for the divided voltage in the external voltage divider

By inserting the above values in equation 3.19, the battery voltage can be found by the result presented in equation 3.20

$$V_{BAT} = V_{IN} \cdot \left(\frac{0.6V}{1 \cdot 2^{10}} \right) \cdot \left(\frac{10k\Omega + 82k\Omega}{10k\Omega} \right) = V_{IN} \cdot 5.3902625 \cdot 10^{-3} \tag{3.20}$$

3.6.2.3 I^{2}C

I^{2}C communication is executed by the use of drivers provided with the Nordic Semiconductor SDK, however the name of the provided I^{2}C examples and functions is Two-Wire Interface (TWI). Initialization is done at startup by declaring which pins to be used for SDA and SCL, as well as other settings as seen in code 3.6.

Code 3.6: I^{2}C Configuration

```
    nrf_drv_twi_config_t twi_config;

    twi_config.sda                = 26;
    twi_config.scl                = 27;
    twi_config.frequency          = NRF_TWI_FREQ_100K;
    twi_config.interrupt_priority = APP_IRQ_PRIORITY_HIGH;
    twi_config.clear_bus_init     = false;

    nrf_drv_twi_init(&m_twi, &twi_config, NULL, NULL);
```

In order to communicate with the I^{2}C slave devices, two functions are used. *nrf_ drv_ twi_tx(ID,ADDR,DATA,LENGTH,NO_ STOP);* is used to write data to the slave device. *ID* selects which I^{2}C connection is used for a given slave. *ADDR* is the main address of the slave with which the master wants to communicate. *DATA* contains the data to be written to the slave. This can either be one byte for selecting a register to be read from the slave, or it can be multiple bytes to write new register values to the slave. *LENGTH* is the transmission length in bytes from the master to the slave. *NO_ STOP* can either be true or false. When false, the connection ends with a stop condition. When true, the stop condition is suppressed so the I^{2}C connection does not go idle.

3.6.2.4 I^2C Scanner

The I^2C scanner for the nRF52840 is designed with the same functionality as the I^2C scanner for the ATmega328p, but with different code. See figure 3.30 in section 3.4.2.2 for a flowchart of the functionality, and appendix B.1.1 for the full code.

3.6.2.5 BME680

The BME680 code for the nRF52840 is based on data from the datasheet, and some function properties from the *ClosedCube_ BME680.h* library are used [43][49]. When initializing the BME680, calibration data needs to be retrieved in order to correctly calibrate the data from the sensor. The calibration data is located in two different memory registers; 0x89 and 0xE1. Combined, these registers contain 36 bytes used for calibration, where 5 are allocated for temperature, 8 for humidity, 16 for pressure and 7 for gas. The full code can be seen in appendix B.4.

IIR filter coefficients are also set when initializing the BME680. The IIR filter helps suppress anomalies and gives a more stable output. There are 8 different levels to which the coefficient can be set. Higher values will increase the strength of the filter, however this also increases the response time of the sensor inputs. The coefficient is set to 4 to limit some anomalies, but still have a low response time.

The BME680 has two different operation modes; sleep mode and forced mode. When the sensor is in sleep, no measurements are performed, and the sensor uses minimal power. In forced mode, a single measurement of the different types of environmental sensors is carried out before returning to sleep. When initializing forced mode, oversampling is also configured to a value from 0 to 5, which allows for more precise measurements, as well as the option to enable or disable measurements of specific environmental data. If oversampling is set to a low value, the measurement will be completed quickly, however may be less accurate as when fewer samples are used. If set to a high value, the accuracy is higher, however each measurement execution takes longer, consuming more energy. The oversampling value is set separately for humidity, temperature and pressure, with the respective values of 2, 1 and 2 for the purpose of the project. Humidity oversampling is written to register 0x72, while temperature and pressure are written to register 0x74. Activating a single sensor read can therefore be written as seen in code 3.7. The *bme680_ask_data()* function tells the BME680 to sense and convert the environmental values. The data can then be retrieved from the corresponding registers when ready. *bme680_ask_data()* retrieves all sensor data except gas. A total of 8 bytes are read consecutively, starting with the pressure registers, followed by temperature and humidity for convenient and efficient retrieval. These values are, in turn, calibrated with calibration data from the initialization in order to get the real results.

Code 3.7: BME680 sensor read activation

```
void bme680_ask_data(void){
  uint8_t forced_reg[2] = {0x72,0x74};
  uint8_t forced_data;
  forced_data = 0x02 & 0b00000111; //hum oversampling
  forced_reg[1] = forced_data;
  nrf_drv_twi_tx(&m_twi,BME_ADDR,forced_reg,2,false);
  forced_data = (0x01 << 5) & 0b11100000; // temp oversampling
  forced_data |= (0x02 << 2) & 0b00011100; // pres oversampling
  forced_data |= 0x01 & 0b00000011; // operation mode
  forced_reg[0] = 0x74;
  forced_reg[1] = forced_data;
  nrf_drv_twi_tx(&m_twi,BME_ADDR,forced_reg,2,false);
}
```

Gas measurement is enabled in a separate register, where heater temperature and duration can be configured. For the purpose of the project, the temperature is configured to 300 $\circ C$ with a period of 200ms. Measuring gas consumes high levels of energy compared to the other measurements, and for that reason, the measuring interval is longer than that of the others. The BME680 outputs resistance values in the range from 0Ω to 500$k\Omega$, where higher values indicate better IAQ.

HDC2010

Even though the code is different for the nRF52840 node compared to the ATmega328p node, the functionality of the code is the same, and the same registers are written to and read from. The explanation of HDC2010 communication can be seen in section 3.4.2, and the full code for nRF52840 in appendix B.1.2.

MAX44009

The default configurations of the MAX44009 are the preferred settings. Therefore, *max_read()* is the only function used for the nRF52840. The *max_read()* function reads two bytes from register 0x03, which contains the necessary data to calculate the LUX value. The full code can be seen in appendix B.1.3.

3.6.2.6 get_sensor_data()

The *get_sensor_data()* function for nRF52840 is similar to that of the ATmega328p, with two main exceptions; battery level sensing, which is described in section 3.6.2.2, and the method for adding data to the radio buffer before transmission. The *radio_mem* buffer is designed to dynamically change with the length of the payload. This functionality can be seen in code 3.8. The first byte, *radio_mem[0]*, is used to set the number of payload bytes. The second byte, *radio_mem[1]*, is designated for the S0 bit, where the LSB of that byte is set as S0. The first payload byte is therefore the third byte, located in *radio_mem[2]*. A variable is set to two and incremented for each new payload byte, and is used in the *radio_mem* element numerator to declare each payload location in *radio_mem*. Lastly, *radio_mem[0]* is set to be equal to $byte - 2$, as this will be the total number of payload bytes. The full code for *get_sensor_data()* is located in appendix B.1.4.

Code 3.8: *radio_mem[x]*

```
uint8_t byte = 2;
radio_mem[byte++] = Payload_data;
radio_mem[byte++] = Payload_data;
.
.
.
radio_mem[byte++] = Payload_data;
radio_mem[0] = byte - 2; // NR payload bytes
radio_mem[1] = 0;        // Unused
```

3.6.2.7 nRF52840 Radio

Radio communication is based on the "radio" example and on the *NRF_RADIO_Type* struct in the nrf52840.h file, which contains the control registers for the radio, both of which can be found in the SDK. The full code for the nRF52840 radio is presented in appendix B.1.6. The following five functions interface directly with the radio by writing to, and reading from, its registers:

- *radio_setup()*

- *send_packet()*
- *start_listening()*
- *stop_radio()*
- *carrier_sense()*

The *radio_setup()* function contains the operation parameters of the radio. Two different packet structures are necessary in order to support both normal and coded communication. The main difference is the preamble which is 1 byte long for $1Mb$ and $2Mb$ proprietary mode, and 10 bytes for $125kb$ BLE coded mode.

Code 3.9 shows the parameters which are configured in *radio_setup()*.

Code 3.9: *radio_setup()* parameters

```
NRF_RADIO->BASE0 = (uint32_t)(base0);
NRF_RADIO->BASE1 = (uint32_t)(base1);
NRF_RADIO->PREFIX0 = (uint32_t)(pre_addr0);
NRF_RADIO->PREFIX1 = (uint32_t)(pre_addr1);
NRF_RADIO->TXADDRESS = 0;
NRF_RADIO->RXADDRESSES = 0b00000001;
NRF_RADIO->PACKETPTR = (uint32_t)&radio_mem;
NRF_RADIO->SHORTS |= RADIO_SHORTS_ADDRESS_RSSISTART_Msk;
NRF_RADIO->FREQUENCY = 7UL;
NRF_RADIO->TXPOWER = powerLVL[currentPL];
```

BASE and *PREFIX* are the address values which are used. Different parts of these addresses are used, which is determined by the values in *TXADDRESS* and *RXADDDRESSES*. *TXADDRESSES* can be values between 0-7, which are the number of channels available. *RXADDDRESSES* consists of 8 bits which can enable or disable listening on the corresponding channel. The LSB is channel 0 and the MSB is channel 7. *PACKETPTR* is a pointer to where the radio will store incoming packets, and also where data that is going to be sent should be located before transmission. *PACKETPTR* stores the pointer to *radio_mem*, which is an *uint8_t* array with 20 elements. *RADIO_SHORTS_ADDRESS_RSSISTART_Msk* is a shortcut operator which takes an RSSI sample on ADDRESS match, which lets the receiver automatically sample RSSI of each incoming packet with address match. *FREQUENCY* takes values between 0 to 100MHz and sets the radio frequency equal to 2400MHz + the number in that register. *TXPOWER* sets the transmitting dBm with values ranging from $-40dBm$ to $+8dBm$. In order to dynamically change this value, an array containing 11 different power levels is created, as shown in code 3.10

Code 3.10: Power level array

```
uint8_t powerLVL[11] = {
    0xD8UL, // -40 dbm
    0xECUL, // -20 dbm
    0xF4UL, // -12 dbm
    0xFCUL, // -4 dbm
    0x0UL,  // 0 dbm
    0x2UL,  // 2 dbm
    0x4UL,  // 3 dbm
    0x5UL,  // 5 dbm
    0x6UL,  // 6 dbm
    0x7UL,  // 7 dbm
    0x8UL   // 8 dbm
};
uint8_t currentPL = 10;
```

TXPOWER is initially configured at +8dbm. It can however be lowered automatically if the sensed RSSI of the receiver is unnecessarily high.

The *send_packet()* function ensures that the receiver is completely turned off, and the transmitter completely turned on before the packet is sent. When the send packet command is sent to the radio, it will check the first byte in *radio_mem* to ascertain the payload length of the transmission before transmitting the correct number of elements.

start_listening() sets the receiver in listening mode. While listening, successfully received packets will be stored in the *radio_mem* array.

stop_radio() is used to turn the radio off. For sensor nodes, this function is called after each transmitted packet, as well as after each received ACK or timeout in order to reduce energy consumption. Turning off the radio before changing the radio register, such as changing the power level, is required.

The *carrier_sense()* function shown in code 3.11 turns on the receiver and initiates *TASKS_EDSTART*, which detects the energy levels on the channel for a period of $128\mu s$. The detected energy levels range from 0 to 255, where 0 is at most $10dBm$ higher than the sensitivity of the receiver. In other words, as the receiver has a sensitivity of $-95dBm$, the sensed value is 0 in the range of $-95dBm$ to $-85dBm$. The result can then be retrieved from EDSAMPLE, where any value above 0 indicates traffic on the channel.

Code 3.11: *carrier_sense()* function

```
bool carrier_sense() {
  NRF_RADIO->TASKS_RXEN = 1U;
  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->TASKS_EDSTART = 1;
  nrf_delay_us(130);
  if (NRF_RADIO->EDSAMPLE < 1) {
    return 1;
  } else {
    return 0;
  }
}
```

3.6.2.8 Star topology

When the sleep timer interrupt wakes the node from sleep, the *wakeup()* function is called, which starts the HFCLK before calling *get_sensor_data()* and *send_data()* as shown in code 3.12. The functionality of *get_sensor_data()* is described in section 3.6.2.6.

Code 3.12: Star *wakeup()* function

```
static void wakeup(void *p_context) {
  UNUSED_PARAMETER(p_context);
  start_HFCLK();
  get_sensor_data();
  send_data();
}
```

The nRF52840 star topology communication is similar to the ATmega328p communication, with the exception of a different dynamic transmission power level adjustment method. The RSSI is transmitted in the ACK message from the gateway, and the *wait_for_ack()* function adjusts the transmission power accordingly. Therefore, the star topology is performed with only three main functions instead of the four in the ATmega328p firmware. The *send_data()* function is used to initiate transmission.

- *carrier_sense()*
- *send_data()*
- *wait_for_ack()*

Figure 3.46 shows the structure of a transmission. The data packet contains the transmitting nodes power level in order to allow for the gateway to respond with an ACK at the same power level, effectively reducing channel interference. Before transmitting, the *carrier_sense()* function is called, which senses the channel and determines if the channel is free to use. After transmitting the packet, the *wait_for_ack()* function is called. When waiting for ACK, the node first calls the *start_listening()* function, causing it to wait for up to 1500μs for an ACK from the gateway with the correct ID. When an ACK is received, the RSSI is automatically stored in NRF_RADIO->RSSISAMPLE, and is used to change the power level. The value is a positive integer in the range of 0 to 128, however the actual number it represents is a negative dBm value of the RSSI. The ACK message contains the RSSI from the packet received at the gateway, and the lowest RSSI value of the messages communicated between the node and gateway is used to adjust the power level. An upper and lower limit is set to $-75dBm$ and $-85dBm$ accordingly in order to maintain a good signal strength while limiting interference. If the RSSI is below $-85dBm$, the power level is increased, unless it is already at the highest power level. If higher than $-75dBm$, the power level is decreased unless it is already at the lowest setting. If no ACK is received from the gateway, the power level is increased.

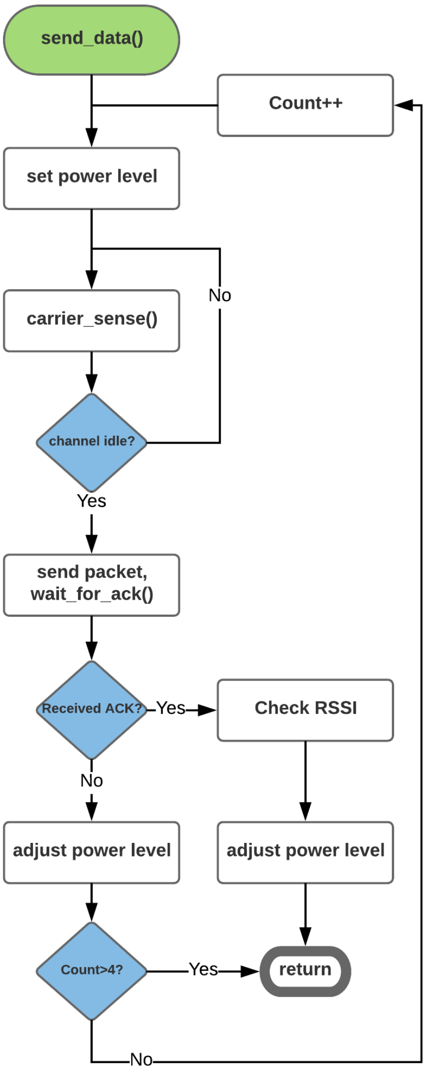


Figure 3.46: Flowchart of *send_data()*

3.6.2.9 Multi-hop topology

When the sleep timer interrupt wakes the node from sleep, the *wakeup()* function shown in code 3.13 is called. The function operates similarly to the main loop of the ATmega328p, as can be seen in section 3.4.2.10. When waking up, the function either sends its own sensor data first before calling *relay_data()*, or only calls *relay_data()*, depending on the *send_counter* variable. If the node receives a response, the function will return 1 and the node will attempt to forward the data towards the gateway. After forwarding the data, the *relay_data()* function is called once more. This process continues until no response is received during *relay_data()*.

Code 3.13: multi-hop *wakeup()* function

```c
void wakeup(void *p_context) {
  UNUSED_PARAMETER(p_context);
  start_HFCLK();
  if ((send_counter >= SEND_INTERVAL) || (my_hops_to_sink > 199)) {
    get_sensor_data();
    wait_to_send();
    while (relay_data()) {
    }
    send_counter = 0;
  } else {
    while (relay_data()) {
    }
  }
  app_timer_start(wakeping_timer, WAKEPING_INTERVAL, NULL);
  send_counter++;
}
bool relay_data() {
  wake_ping();
  if (wait_for_response()) {
    wait_to_send();
    return 1;
  }
  return 0;
}
```

The multi-hop functionality is for the most part the same for nRF52840 as it is for ATmega328p. The protocol is explained in section 3.3.3, and the implementation for ATmega328p in section 3.4.2.10. The nRF52840 radio does however have additional features which improves the system. For communication, three pipes are used for each node; wake pipe, next hop pipe and node ID pipe. Wake pipe is used for transmitting and receiving wake pings, next hop pipe is used for sending data to a specific node and node ID pipe is used for receiving packets sent specifically to itself.Only one pipe is open at a time in order to limit any undesired traffic when sending or receiving. The three pipes represent three different addresses which the node will listen for, where wake pipe address is identical for all nodes and node ID pipe address is unique for each node. The next hop pipe address at the sender will correspond with the node ID pipe address of the receiver, and can initially only be set after a wake ping is received.

When waiting to transmit data, the node will listen on the wake pipe and wait for wake pings with hops to gateway lower than its own. If such a message is received, the node will automatically set its next hop pipe to to match the ID of the wake ping, before sending the data message. Immediately after, it will start listening on its node ID pipe for an ACK. The benefit of using this method is that the nRF52840 radio has a function that lets the receiving node know when a message with a recognized address is incoming. This means that the radio can identify an incoming message immediately after the preamble and address part of the packet, even before the payload is received. This is beneficial because while waiting for an ACK, or more importantly a response, where the payload contains sensor data, the duration the node has to wait can be greatly reduced. The process of waiting for ACK is shown in figure 3.47. Waiting for ACK is set to $800\mu s$ and wait for response is set to $1000\mu s$. Wait for response is $200\mu s$ longer because a random backoff from to $0 - 80\mu s$ and a carrier sense of $128\mu s$ is performed prior to sending. If an address is recognized withing these durations, it will wait for the whole packet to be received. The code for nRF52840 multi-hop can be found in appendix B.2.

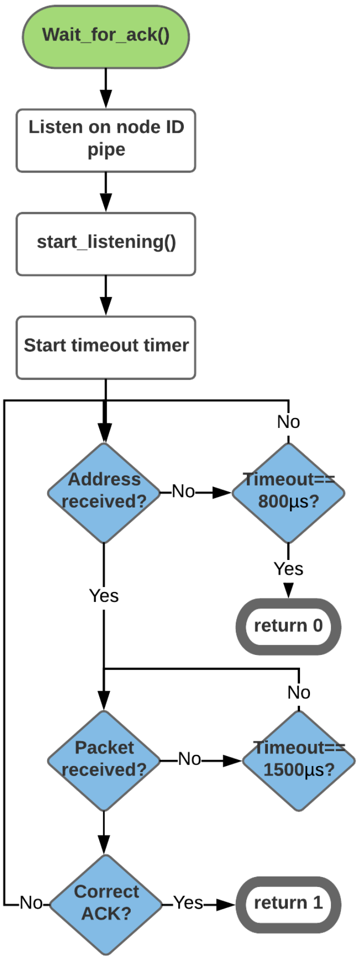


Figure 3.47: Flowchart of *wait_for_ack()*

3.7 nRF52840 Gateway

The nRF52840 gateway utilizes an nRF52840 MCU along with the ESP32. The nRF52840 communicates wirelessly with the nodes and sends relevant data to the ESP32 via UART. The ESP32 forwards the data via Wi-Fi to an MQTT broker.

3.8 nRF52840 Gateway Hardware

Figure 3.48 shows a PCB designed for the gateway. The ESP32 is programmed by using a USB to Transistor-Transistor Logic converter, while the BMD-340 can be programmed by using either USB or J-Link. The supporting components are used according to the datasheets of the ESP32 and BMD-340 [56] [53]. The antennas are placed on opposite edges of the PCB in order to minimize interference between them. The gateway is supplied with 3.3V. All schematics for the nRF52840 gateway, as well as the PCB, can be found in appendix D.4.

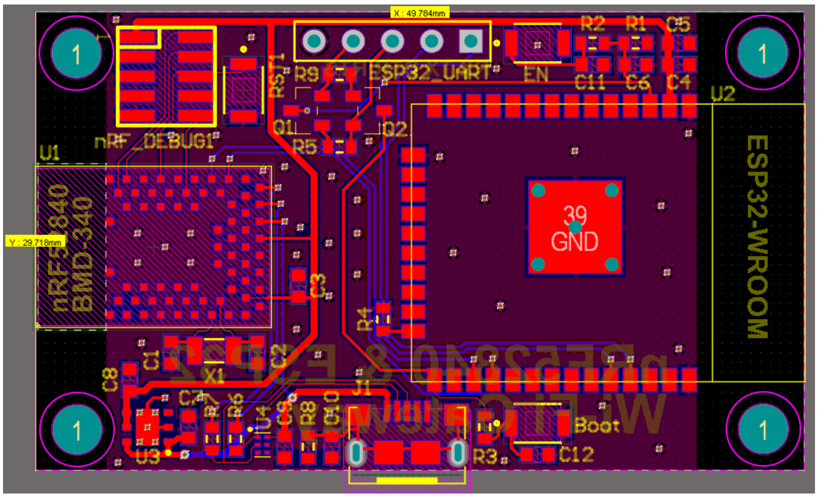


Figure 3.48: nRF52840 gateway PCB

3.9 nRF52840 Gateway Firmware

nRF52840

A nRF52840 is used with the ESP32 to receive data from multi-hop and star nodes. Its main function is to relay data using UART to the ESP32, which will publish this data to MQTT. When used in star topology, it will only wait for data packets, and send an ACK back to the node. When the packet is received it will first check if it is an already received packet, and discard it if is a duplicate. Either way an ACK is sent back to the node with the same power level. The power level of the node is sent with the data packet, which is used by the nRF52840 gateway to adjust its own. The ACK message contains the senders ID and the RSSI of the received data packet. The RSSI is used by the node to adjust its power level to reduce interference. Finally the data is sent through the UART to the ESP32. When used with multi-hop topology, a wake ping is sent every two seconds so neighbouring nodes can find it.

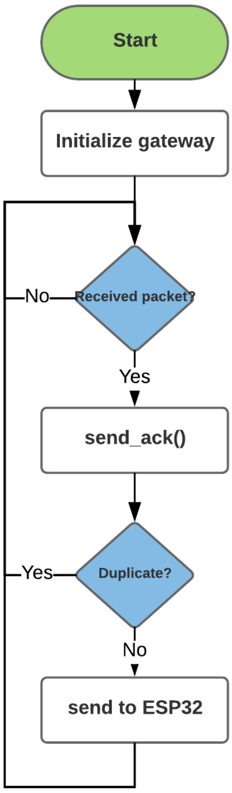


Figure 3.49: Flowchart of nRF52840 gateway

ESP32

The ESP32 gateway for the nRF52840 system is similar to the ESP32 gateway for the ATmega328p system. Instead of reading from the nRF24l01+ radio, the ESP32 reads from UART connected by wire to the nRF52840. Full code for the ESP32 nRF52840 gateway can be found in appendix A.4.

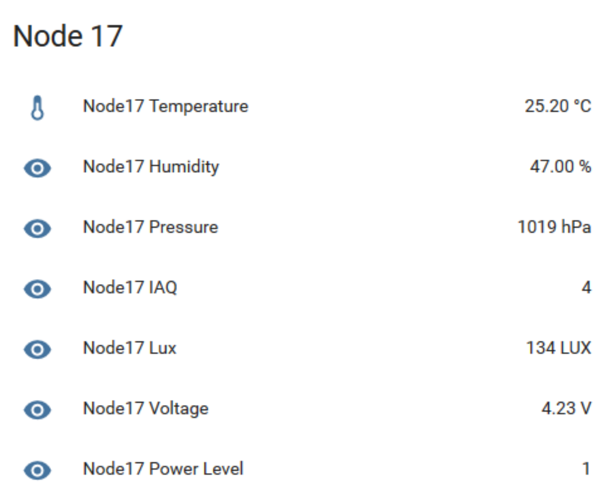
Code 3.14: Main loop

```
void loop()
{
  if ((Serial2.available() > 7)) {
    if (Serial2.read() == 155) {
      data[0] = Serial2.read();
      for (int i = 1; i < data[0] + 2; i++) {
        data[i] = Serial2.read();
      }
      while (Serial2.available()) {
        Serial2.read();
      }
      publish_mqtt();
      for (int i = 0; i < 20; i++) {
        data[i] = 0;
      }
    }
  }
}
```

3.10 IoT Server

In order to gather, log and display the data from the sensor nodes,
a Raspberry Pi is used along with Hass.io and Mosquitto. Hass.io is powered by Home Assistant, an open source software developed for simple home automation. For the purpose of the project, however, only the integrated MQTT support is utilized, which requires minimal setup. Hass.io can subscribe to any topic on the MQTT broker and present the received data in a graphical interface, as well as create graphs, both of which are utilized throughout the development and testing of the system.

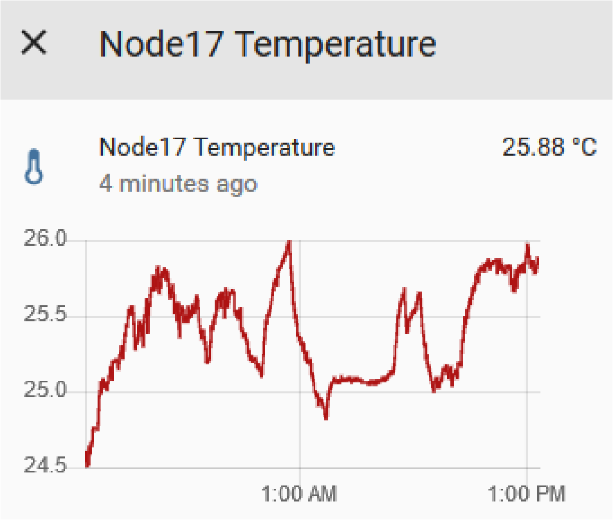
Figure 3.50 depicts all received data from Node 17, which is an ATmega328p node with BME680, while figure 3.51 shows the graphical representation of its temperature reading history. A power level of 1 indicates *LOW* on the nRF24l01+PA+LNA.



Node 17

Node17 Temperature	25.20 °C
Node17 Humidity	47.00 %
Node17 Pressure	1019 hPa
Node17 IAQ	4
Node17 Lux	134 LUX
Node17 Voltage	4.23 V
Node17 Power Level	1

Figure 3.50: Node 17 group in Home Assistant



Node17 Temperature

Node17 Temperature 25.88 °C
4 minutes ago

Figure 3.51: Temperature history of node 17 in Home Assistant

Additionally, a separate Raspberry Pi is configured with an MQTT broker, namely Mosquitto, in addition to a python script which logs published data in all topics.

Chapter 4

Testing

4.1 Energy Harvesting Testing

The energy harvesting circuit is tested with both configurations described in section 3.1.2, with the 120mAh battery and the 5F supercapacitor. Both solar panels described in section 3.1.1 are tested with the energy harvesting circuit. The tested system is placed in the same location throughout all the tests, in a room with multiple windows and a desk LED light on at all times. The energy harvesting tests are conducted with the Mooshimeter, which can measure and log the voltage on and current to the energy storage element. The Mooshimeter a highly portable app controlled multimeter with BLE.

Figure 4.1 shows the room in which the nodes are tested, while figure 4.2 shows the PV panels while undergoing testing.

Figure 4.1: Test location

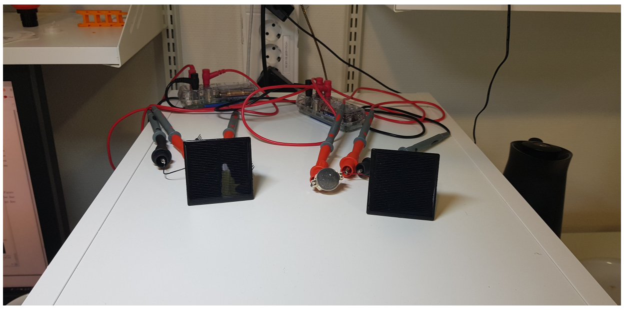


Figure 4.2: PV panel testbed

4.1.1 Battery and Monocrystalline Panel

The 120mAh Li-Ion battery is tested along with the 0.36W monocrystalline panel. Figure 4.3 depicts the resulting values over a period of approximately 2.5 days, where the blue line represents power in μW with a running average of 20 measurements, and the orange line voltage. The average power by which the battery was charged during the test was $941.94\mu W$.

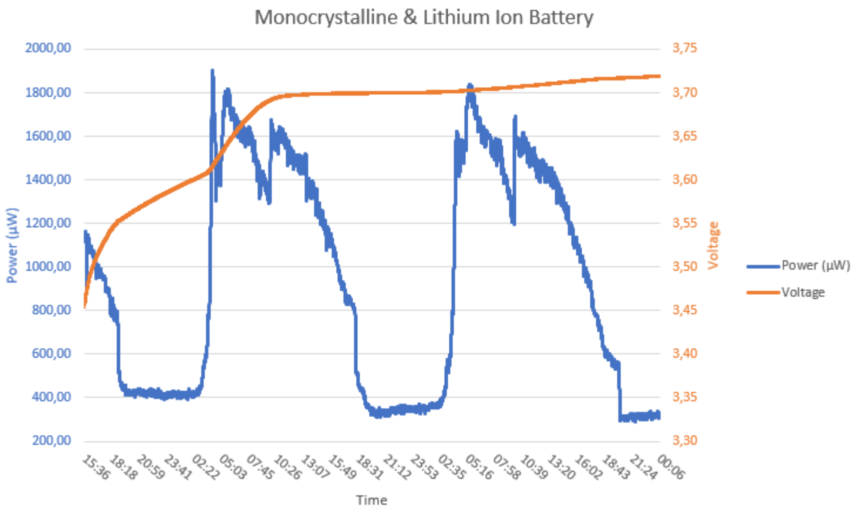


Figure 4.3: Charging Li-Ion battery with monocrystalline panel

4.1.2 Battery and Polycrystalline Panel

The $120mAh$ Li-Ion battery is tested along with the $0.15W$ polycrystalline panel. Figure 4.4 depicts the resulting values over a period of approximately 9 days, where the blue line represents power in μW with a running average of 20 measurements, and the orange line voltage. The average power by which the battery was charged during the test was $213.96\mu W$.

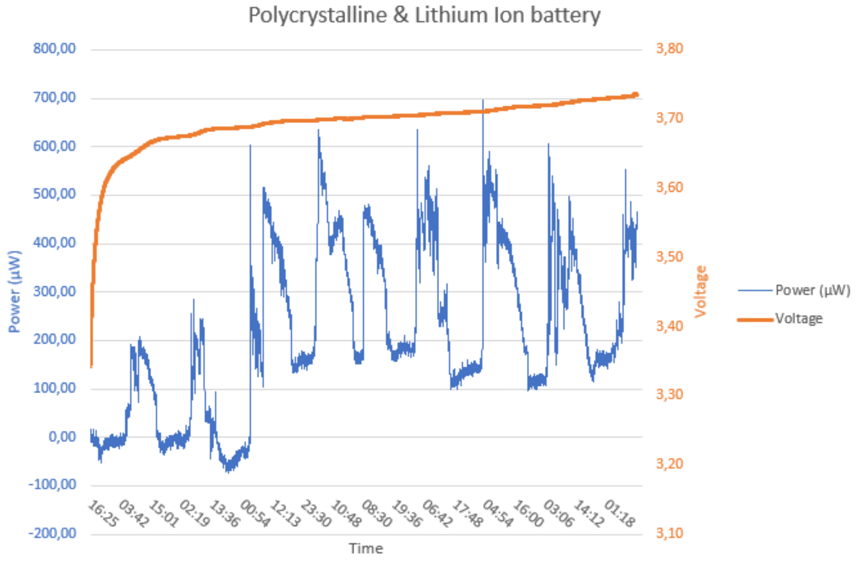


Figure 4.4: Charging Li-Ion battery with polycrystalline panel

4.1.3 Supercapacitor and Monocrystalline Panel

The $5F$ supercapacitor was tested along with the $0.36W$ monocrystalline panel over a period of approximately 9 days. Figure 4.5 depicts the charging from a near depleted state, where the blue line represents power in μW with a running average of 20 measurements, and the orange line voltage. As can be seen in the figure, the BQ25570 is unable to charge a near depleted supercapacitor of $5F$ as the voltage nears $1.3V$. This is due to the self discharge and to the initial charge issue of the BQ25570, as explained in section 3.1.4, where a very low charging current is obtained. The average charging power was $11.34\mu W$

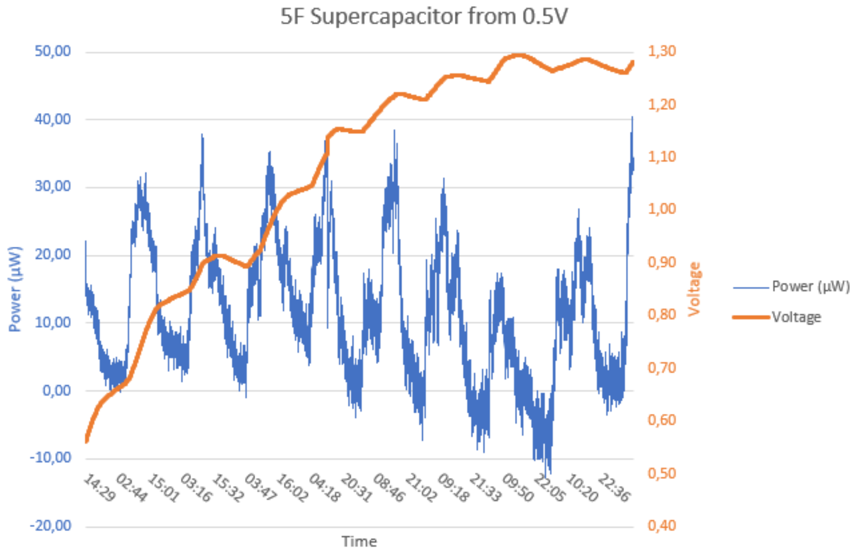


Figure 4.5: Charging a $5F$ supercapacitor from $0.5V$ with monocrystalline panel

After cancelling the initial charge test, the supercapacitor was pre-charged to approximately $2.8V$ before continuing the test. Figure 4.6 shows the period from $2.8V$ to full charge at $5.28V$. The blue line represents power in μW, while the orange represents the supercapacitor voltage. As can be seen, the test started at 11:35, and the supercapacitor reached the programmed maximum voltage at 05:45 the next day, resulting in a total charging period of 18 hours and 10 minutes. The average charging power was $1249.5\mu W$ between $2.8V$ and $5.28V$.

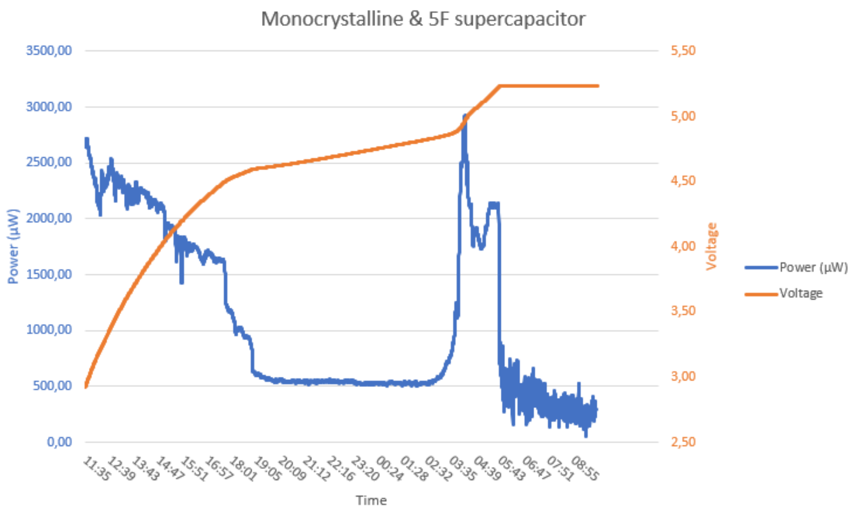


Figure 4.6: Charging a $5F$ supercapacitor from $2.8V$ with monocrystalline panel

4.1.4 Supercapacitor and Polycrystalline Panel

As the BQ25570 was unable to charge the supercapacitor from the depleted stage, as described in section 4.1.3, the supercap was pre-charged to approximately $2.7V$ for before testing with the $0.15W$ polycrystalline panel.

Figure 4.7 shows the period from $2.7V$ to full charge at $5.281V$. The blue line represents power in μW, while the orange represents the supercapacitor voltage. Due to unknown errors in the Mooshimeter, a gap of 5 hours occured on the second day of testing. The results are therefore not completely accurate, however provides an indication of the performance. As can be seen, the test started at 13:45, and the supercapacitor reached the programmed maximum voltage after 56 hours. The average charging power was $303.9\mu W$.

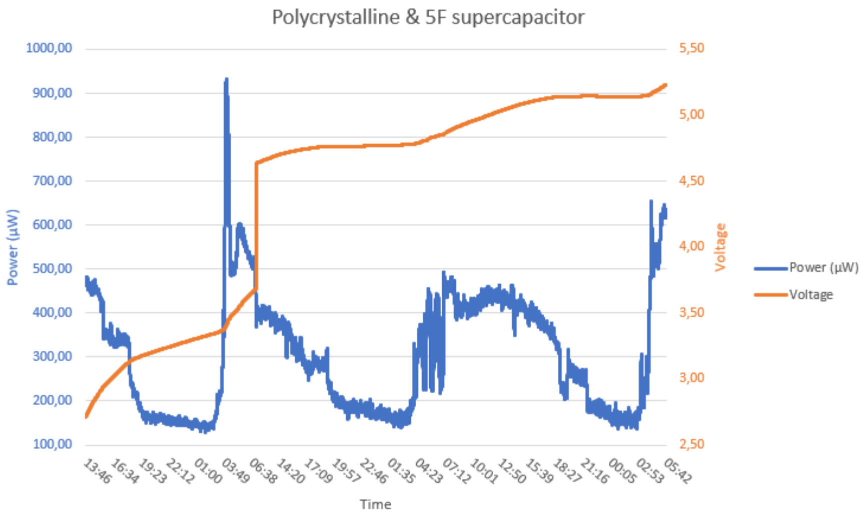


Figure 4.7: Charging a $5F$ supercapacitor from $2.7V$ with polycrystalline panel

4.1.5 Low Light Energy Harvesting

A test was conducted with the $120mAh$ battery as storage medium with both the $0.15W$ polycrystalline and the $0.36W$ monocrystalline panels in a room with no windows to the outside and overall low lighting. Figure 4.8 depicts the room in which the test was conducted, and figure 4.9 shows the EH PCBs while undergoing testing.

Figure 4.8: Dark room test location

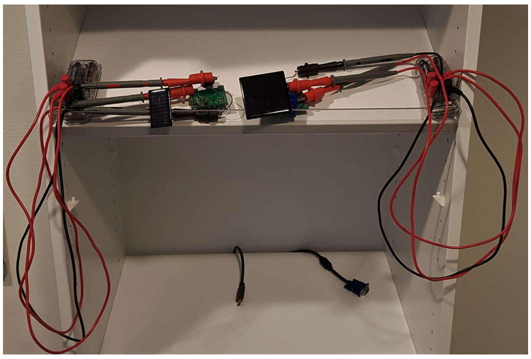


Figure 4.9: Dark room testbed

Figure 4.10 depicts the test conducted with the monocrystalline panel and battery in low light conditions over a period of approximately $24h$. The average power obtained was $212\mu W$. A slight battery voltage increase is observed from $3.421V$ to $3.469V$.

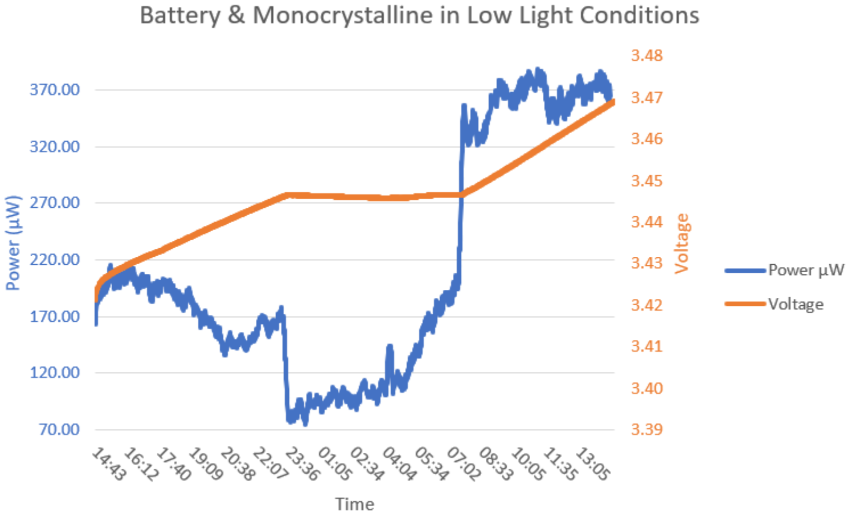


Figure 4.10: Monocrystalline and battery in low light conditions

As for the test with polycrystalline and battery in low light conditions, the power measurement was unstable as a cause of too low values for the Mooshimeter. Therefore, only the battery voltage is presented in figure 4.11. The increase in voltage during the measurement period of $24h$ was from 3.442 to 3.447

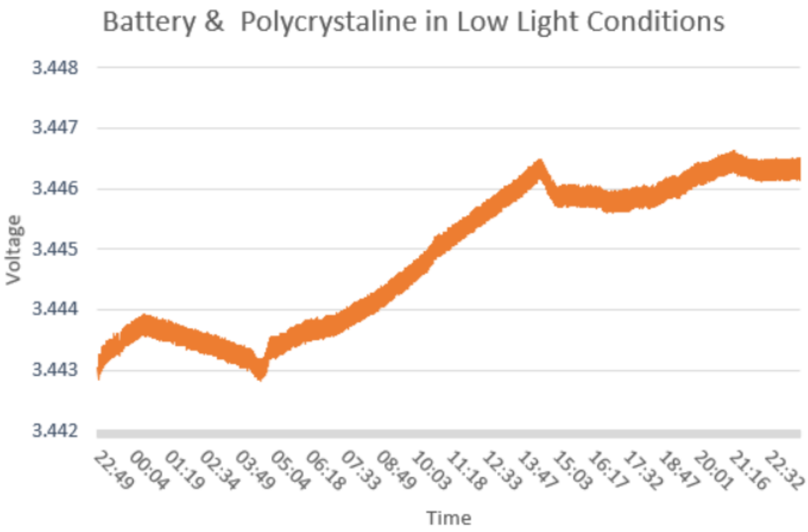


Figure 4.11: Polycrystalline and battery in low light conditions

4.1.6 Comparison of Polycrystalline and Monocrystalline

During the battery testing in a well-lit room, the polycrystalline panel provided an average of $231.96\mu W$, while the monocrystalline panel provided $941.94\mu W$. Equation 4.1 is used to calculate the efficiency of the polycrystalline PV panel relative to the monocrystalline panel.

$$Efficiency = \left(\frac{P[avg]_{poly}}{P[rated]_{poly}}\right) \cdot \left(\frac{P[rated]_{mono}}{P[avg]_{mono}}\right) \cdot 100\% \tag{4.1}$$

Where:

- Efficiency is the efficiency of the polycrystalline panel relative to the monocrystalline panel
- $P[avg]_{poly}$ is the average obtained power from the polycrystalline panel
- $P[rated]_{poly}$ is the rated maximum power of the polycrystalline panel
- $P[avg]_{mono}$ is the average obtained power from the monocrystalline panel
- $P[rated]_{mono}$ is the rated maximum power of the monocrystalline panel

By inserting the values obtained during the tests, the efficiency can be found as shown in equation 4.2

$$Efficiency = \left(\frac{231.96\mu W}{0.15W}\right) \cdot \left(\frac{0.36W}{941.94\mu W}\right) \cdot 100\% = 54.5\% \tag{4.2}$$

During the testing of the supercapacitor in a well-lit room, the average power obtained from the polycrystalline panel was $303.9\mu W$, while the monocrystalline panel provided $1249.5\mu W$.

The relative efficiency is calculated in equation 4.3.

$$Efficiency = \left(\frac{303.9\mu W}{0.15W}\right) \cdot \left(\frac{0.36W}{1249.5\mu W}\right) \cdot 100\% = 58.5\% \tag{4.3}$$

4.2 ATmega328p Node Test

4.2.1 Energy Consumption

The measurements of energy consumption in the ATmega328p nodes are conducted by the use of the Keysight InfiniiVision DSOX2002A digital oscilloscope [36]. In order to obtain a the current values, the voltage over a 5Ω resistor is measured in series with the system. Therefore, the current is found by calculating $I = V_{measured}/5\Omega$.

4.2.2 Sleep Consumption

Figure 4.12 shows the sleep current of the ATmega328p node with BME680. The sleep current is measured with the precision electrometer Keysight B2950A over 20s, during which the average current was 5.47μA, and the average power 18.1μW. The peaks observed in the sleep measurements are caused by the MCU waking up after a set sleep interval of up to 8s, only to return to sleep.

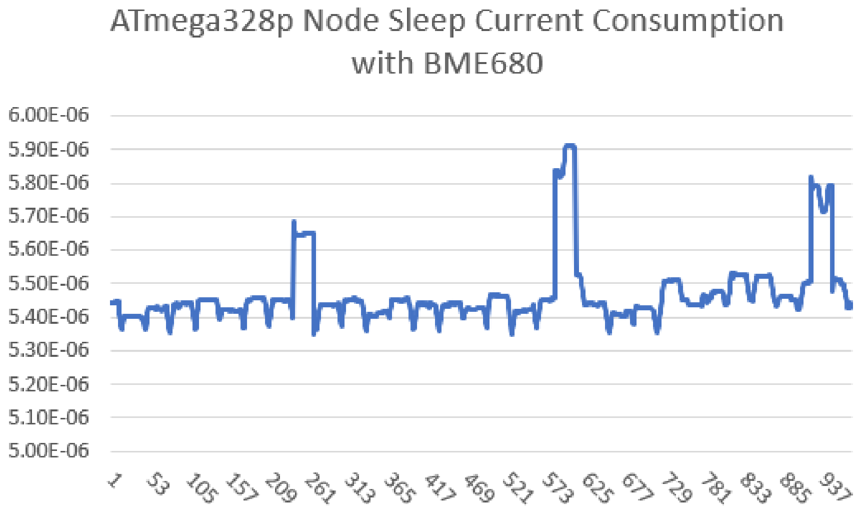


Figure 4.12: ATmega328p sleep current measurement with BME680

Figure 4.13 shows the sleep current of the ATmega328p node with HDC2010. The average current was 5.4μA, and the average power 17.8μW

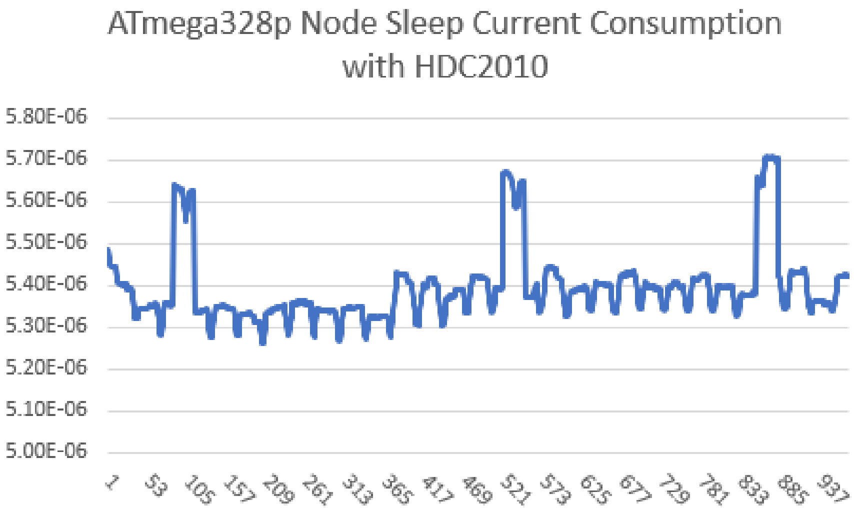
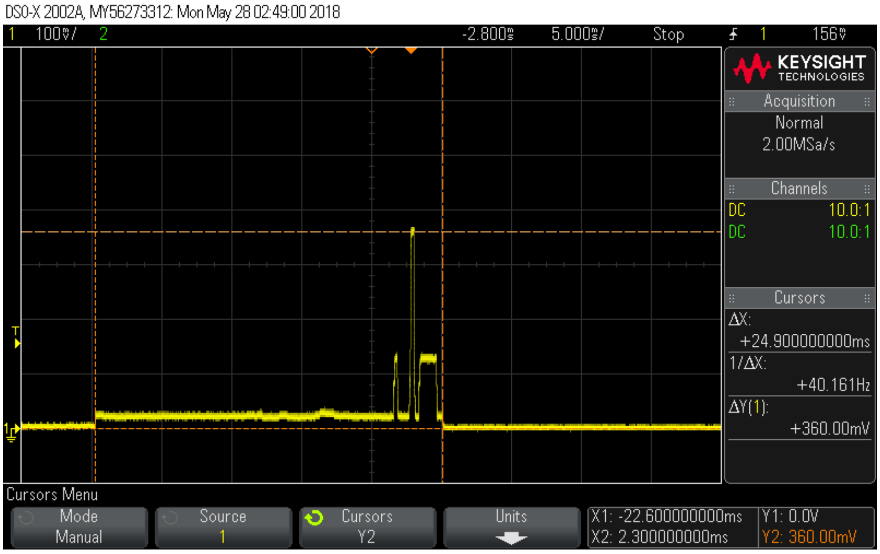


Figure 4.13: ATmega328p sleep current measurement with HDC2010

4.2.2.1 Star

Figure 4.14 shows the measurement of the full on-time of one transmission from the ATmega328p node in star topology. Cursors X1 and X2 indicate the total on-time of the node, while Y1 and Y2 indicates the peak voltage measured over the 5Ω resistor. As seen in the figure, the total on-time of the node is 24.9ms, which consists of a 22.82ms sensor read, 420μs transmission and 1.66ms carrier sensing and ACK receival period.


DSO-X 2002A, MY56273312: Mon May 28 02:49:00 2018

Acquisition: Normal, 2.00MSa/s

Channels: DC 10.0:1, DC 10.0:1

Cursors
ΔX: +24.900000000ms
1/ΔX: +40.161Hz
ΔY(1): +360.00mV

Cursors Menu: Mode Manual; Source 1; Cursors Y2

X1: -22.600000000ms | Y1: 0.0V
X2: 2.300000000ms | Y2: 360.00mV

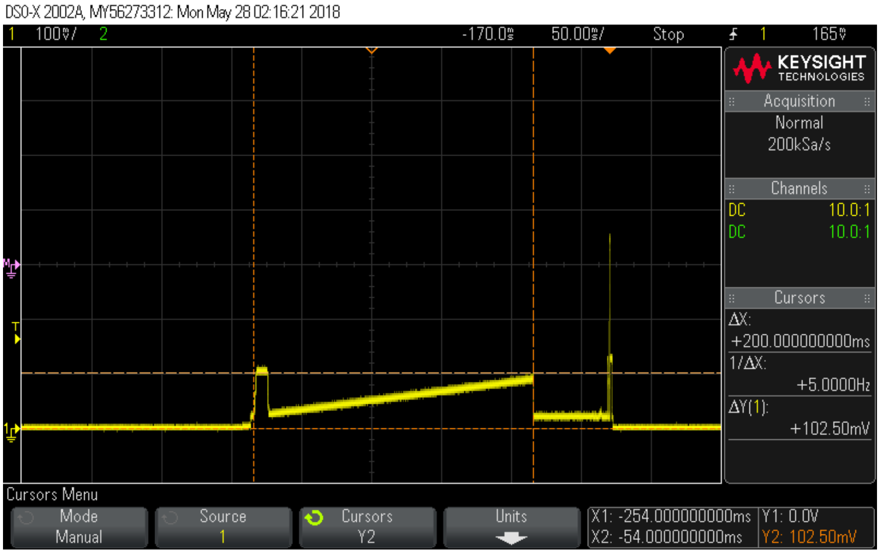
Figure 4.14: ATmega328p oscilloscope measurement in star topology with LOW transmission level and BME680

Table 4.1 presents the current and power consumption based on figure 4.14, where power is calculated based on 3.3V operation.

Table 4.1: ATmega328p consumption with LOW and BME680 in star topology

	Period	**Measured Voltage**	**Current**	**Power**
Sensor read	$22.82ms$	$21mV$	$4.2mA$	$13.86mW$
Transmission	$420\mu s$	$360mV$	$72mA$	$237.6mW$
ACK	$1.66ms$	$128mV$	$25.6mA$	$84.48mW$

Every 10th interval, the IAQ is read from the BME680. Prior to each IAQ reading, the internal heater in the sensor is on for approximately $200ms$. This process is shown in figure 4.15.



DSO-X 2002A, MY56273312: Mon May 28 02:16:21 2018

Acquisition: Normal, 200kSa/s

Channels: DC 10.0:1, DC 10.0:1

Cursors: ΔX: +200.000000000ms; 1/ΔX: +5.0000Hz; ΔY(1): +102.50mV

Cursors Menu: Mode Manual; Source 1; Cursors Y2; Units

X1: -254.000000000ms; X2: -54.000000000ms; Y1: 0.0V; Y2: 102.50mV

Figure 4.15: ATmega328p oscilloscope measurement of the BME680 heater

As seen in figure 4.15, a short peak voltage of $102.5mV$ is measured with a period of $10ms$, followed by a linear increase from $27.5mV$ to $87.5mV$ over a period of $193ms$. In order to calculate the average measured voltage during the linear increase, equation 4.4 is used.

$$V_{heater\_ramp\_avg} = \frac{1}{193ms} \cdot \int_0^{193ms} \left(\frac{87.5mV - 27.5mV}{193ms} \cdot x + 27.5mV \right) dx$$

$$V_{heater\_ramp\_avg} = \frac{1}{193ms} \cdot \left[\frac{60mV \cdot x^2}{193ms \cdot 2} + 27.5mV \cdot x \right]_0^{193ms} = 57.5mV \tag{4.4}$$

Equation 4.5 finds the average measured voltage over the entire $203ms$ period.

$$V_{heater\_avg} = \frac{102.5 \cdot 10ms + 57.5mV \cdot 193ms}{10ms + 193ms} = 59.7mV \tag{4.5}$$

Following the result from equation 4.5, the average current of the heater is $I = 59.7mV/5\Omega =$

$11.94mA$, and the average power $P = 11.94mA \cdot 3.3V = 39.4mW$ for a period of $203ms$.

Equation 4.6 is used to calculate the average power of the entire node while active.

$$P_{active} = \frac{P_{sensor} \cdot T_{sensor} + P_{tx} \cdot T_{tx} + P_{ACK} \cdot T_{ACK} + P_{BM680\_heater} \cdot T_{BME680\_heater}}{T_{active}} \tag{4.6}$$

Where:

- P_{active} is the average power during on-state of the node
- P_{sensor} is the power consumed during sensor reading, here $13.86mW$
- T_{sensor} is the period for sensor readings, here $22.82ms$
- P_{tx} is the power consumed during transmission, here $237.6mW$
- T_{tx} is the period for transmission, here $42 + \mu s$
- P_{ACK} is the power consumed during ACK receival, here $84.48mW$
- T_{ACK} is the period for ACK receival, here $1.66ms$
- P_{BME680_heater} is the power consumed by the BME680 heater, here $39.4mW$
- T_{BME680_heater} is the period for the BME680 heater. The period is $203ms$, however as this only occurs on every 10th transmission interval, the value used for calculating P_{active} is $203ms/10 = 20.3ms$.

By inserting the above values in equation 4.6, the average power consumption in awake state is found in equation 4.7.

$$\begin{aligned} P_{active} &= \frac{13.86mW \cdot 22.82ms + 237.6mW \cdot 420\mu s + 84.48mW \cdot 1.66ms + 39.4mW \cdot 20.3ms}{45.2ms} \\ &= 30mW \end{aligned} \tag{4.7}$$

Equation 4.8 presents the method for calculating the average power consumption of the entire node with different transmission intervals.

$$P_{average} = \frac{P_{active} \cdot T_{active} + P_{sleep} \cdot T_{sleep}}{T_{active} + T_{sleep}} = \frac{30mW \cdot 45.2ms + 18.1\mu W \cdot T_{sleep}}{45.2ms + T_{sleep}} \tag{4.8}$$

By ranging T_{sleep} from $10s$ to $300s$, a graph is plotted as shown in figure 4.16. At $55s$ interval, the power consumption is $42.7\mu W$, while it goes below $20\mu W$ at $714s$. The graph presented is based on LOW transmission power, as well as the use of BME680 while reading temperature, RH and pressure on every interval, and IAQ on every 10th interval.

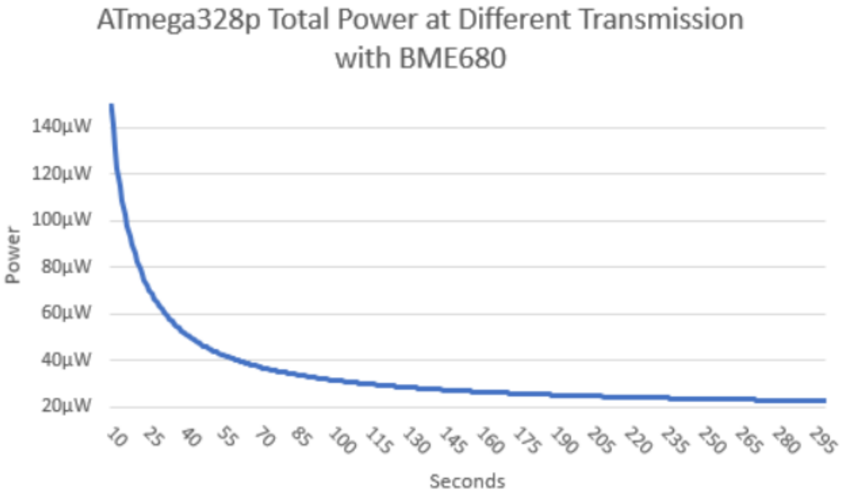
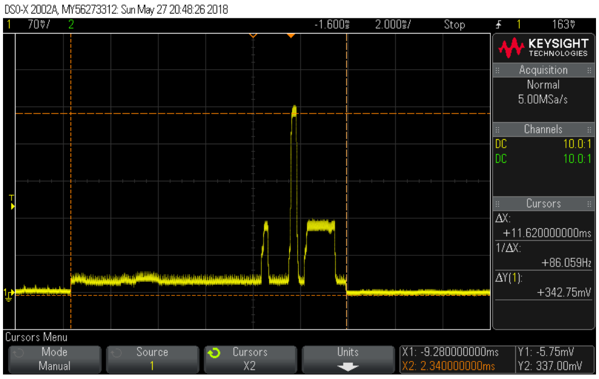


Figure 4.16: Average power consumption of ATmega328p in star topology with different transmission intervals

Figure 4.17 depicts the measurement of one transmission interval with HDC2010 in star topology and a transmission power level of *LOW*. Based on the measurements, the graph in figure 4.18 is plotted, where an average power of $24.8\mu W$ is obtained at a $55s$ transmission interval


DSO-X 2002A, MY56273312: Sun May 27 20:48:26 2018

1 70mV/ 2 -1.600ms 2.000ms/ Stop 1 163mV

Acquisition
Normal
5.00MSa/s

Channels
DC 10.0:1
DC 10.0:1

Cursors
ΔX:
+11.620000000ms
1/ΔX:
+86.059Hz
ΔY(1):
+342.75mV

Cursors Menu
Mode Manual | Source 1 | Cursors X2 | Units
X1: -9.280000000ms Y1: -5.75mV
X2: 2.340000000ms Y2: 337.00mV

Figure 4.17: ATmega328p oscilloscope measurement with HDC2010 and LOW transmission power level

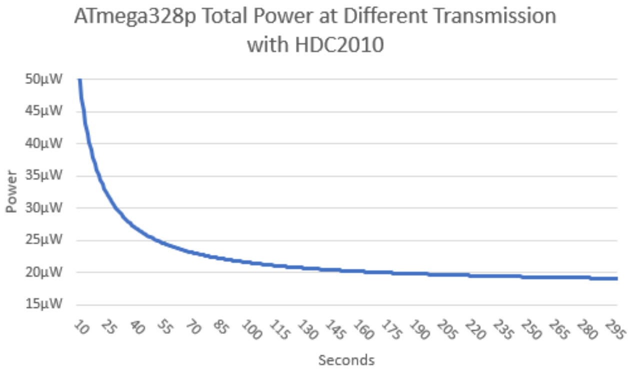
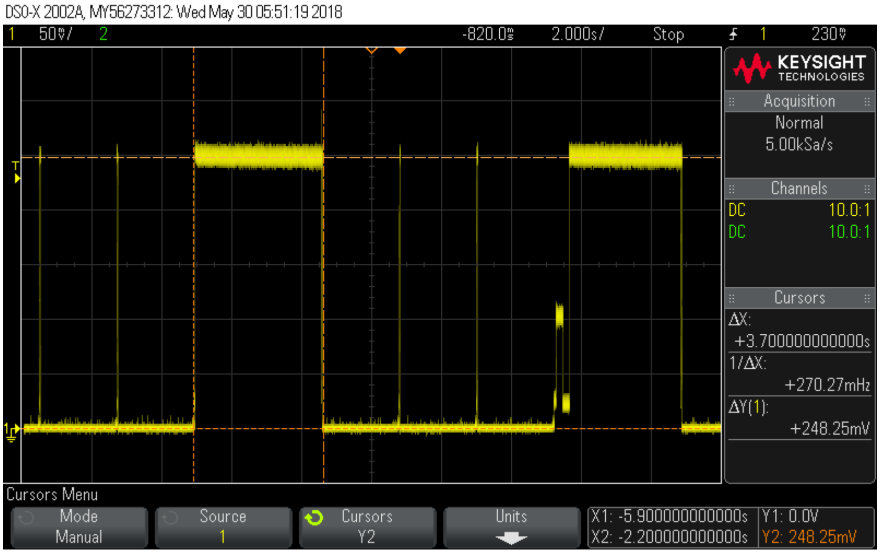


Figure 4.18: Average power consumption of ATmega328p with *LOW* transmission power and HDC2010

4.2.2.2 Multi-Hop

Figure 4.19 depicts an oscilloscope measurement of the general functionality with $2s/div$ horizontally and $50mV/div$ vertically. As can be seen, a wake ping is broadcast every $2s$ during testing, and the nodes data is transmitted on every 3rd wakeup.



DSO-X 2002A, MY56273312: Wed May 30 05:51:19 2018

1 50mV/ 2 -820.0ms 2.000s/ Stop 1 230mV

Acquisition: Normal 5.00kSa/s

Channels: DC 10.0:1, DC 10.0:1

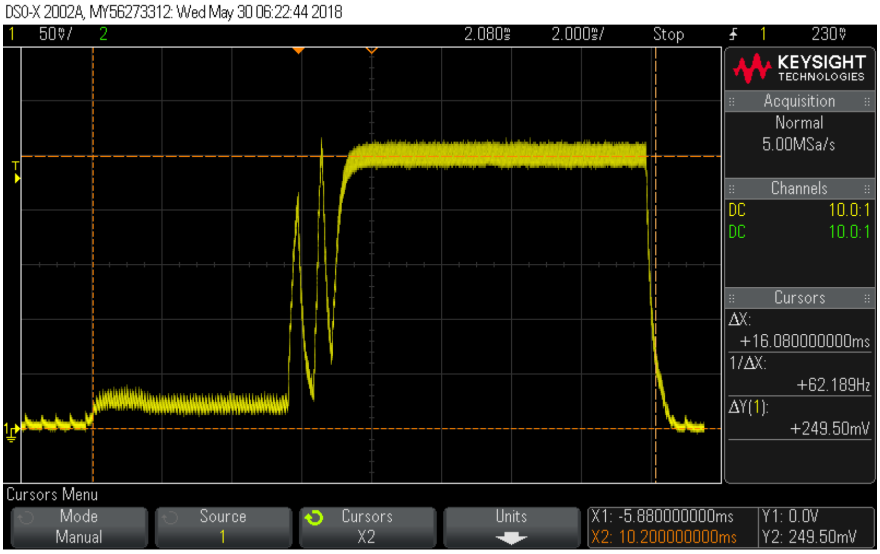
Cursors
ΔX: +3.700000000000s
1/ΔX: +270.27mHz
ΔY(1): +248.25mV

Cursors Menu
Mode: Manual
Source: 1
Cursors: Y2
Units

X1: -5.900000000000s
X2: -2.200000000000s
Y1: 0.0V
Y2: 248.25mV

Figure 4.19: Overview of the multi-hop functionality

Figure 4.20 shows a single wake ping, with the subsequent $10ms$ period of waiting for data. The total on-time of the node was measured at $16.8ms$



DSO-X 2002A, MY56273312: Wed May 30 06:22:44 2018

1 50mV/ 2 2.080ms 2.000ms/ Stop 1 230mV

Acquisition: Normal 5.00MSa/s

Channels: DC 10.0:1, DC 10.0:1

Cursors
ΔX: +16.080000000ms
1/ΔX: +62.189Hz
ΔY(1): +249.50mV

Cursors Menu
Mode: Manual
Source: 1
Cursors: X2
Units

X1: -5.880000000ms
X2: 10.200000000ms
Y1: 0.0V
Y2: 249.50mV

Figure 4.20: Oscilloscope measurement of ATmega328p multi-hop wake ping

Table 4.2 outlines the energy consuming elements of one wake ping. When using a wake ping period of $2s$ and disregarding all other elements of the multi-hop protocol such as data transmission and relaying, the average power consumption is $861\mu W$.

Table 4.2: ATmega328p wake ping measurements

	Period	**Measured Voltage**	**Current**	**Power**
MCU on	$7.16ms$	$23.5mV$	$4.7mA$	$15.51mW$
Carrier sense	$360\mu s$	$204.5mV$	$40.9mA$	$135mW$
Wake ping transmit	$280\mu s$	$248.5mV$	$49.7mA$	$164mW$
Wait for data	$9ms$	$249.5mV$	$49.9mA$	$164.7mW$

Figure 4.21 depicts a transmission of the nodes sensor data where the total measured period was $1.61s$. Figure 4.22 shows the end of figure 4.21 with a lower ms/div in order to present the transmission of data as well as the wake ping and wait for data period. It is worth noting that even though the wait to send period is $1.57s$ in the measurement, the worst case scenario is a period of $2s$.

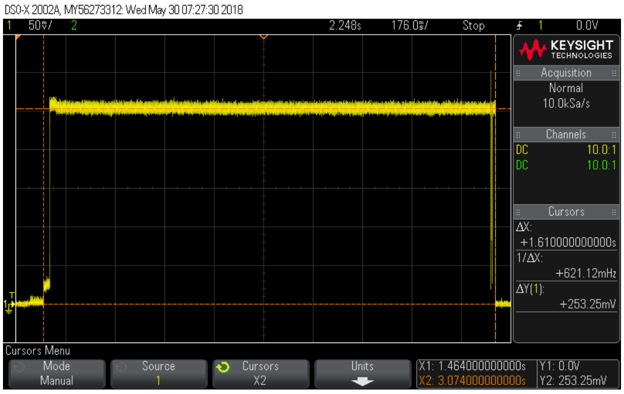


Figure 4.21: Oscilloscope measurement of ATmega328p multi-hop with sensor read, transmit and wake ping

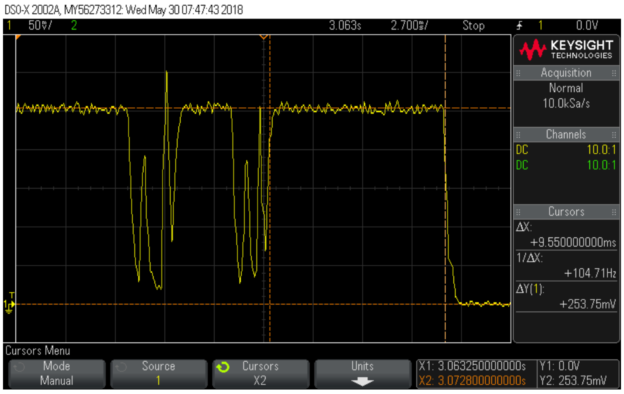


Figure 4.22: Figure 4.21 zoomed

Table 4.3 outlines the consumption and period of each element in figures 4.21 and 4.22.

Table 4.3: ATmega328p read and transmit measurements

	Period	**Measured Voltage**	**Current**	**Power**
MCU on	$25.8ms$	$25mV$	$5mA$	$1.65mW$
Wait to send	$1.57s$	$253.3mV$	$50.7mA$	$167mW$
Carrier sense x2	$600\mu s$	$204.5mV$	$40.9mA$	$135mW$
Transmit data	$300\mu s$	$285mV$	$57mA$	$188mW$
Wait for ACK	$3ms$	$254mV$	$50.8mA$	$168mW$
Wake ping transmit	$280\mu s$	$254mV$	$50.8mA$	$168mW$
Wait for data	$9.6ms$	$254mV$	$50.8mA$	$168mW$

Figure 4.23 shows the measurement of a data relay. The depicted measurement is near worst case, where the wait to send period is equal to $1.918s$

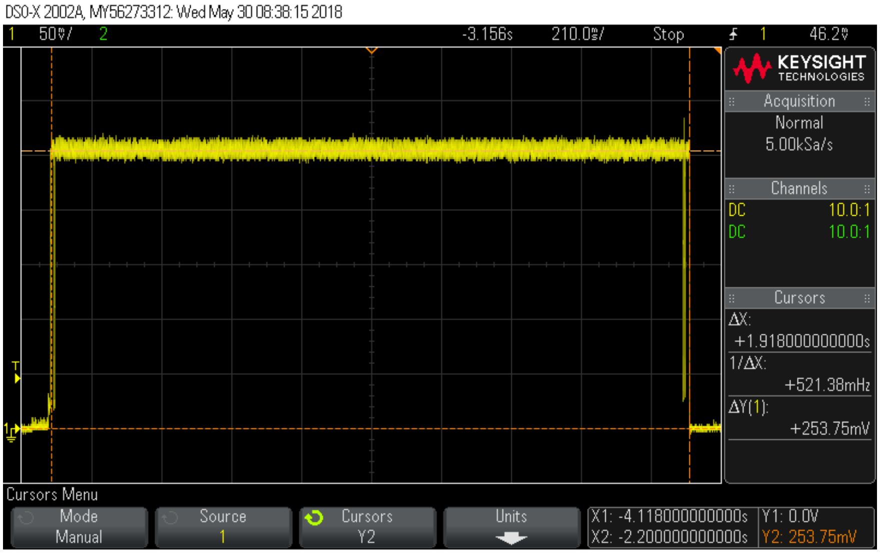


Figure 4.23: Oscilloscope measurement of ATmega328p multi-hop while relaying data

Table 4.4 outlines the values measured in figure 4.23.

Table 4.4: ATmega328p read and transmit measurements

	Period	**Measured Voltage**	**Current**	**Power**
MCU on	$7.16ms$	$23.5mV$	$4.7mA$	$15.51mW$
Carrier sense x2	$600\mu s$	$204.5mV$	$40.9mA$	$135mW$
Wake ping transmit	$280\mu s$	$254mV$	$50.8mA$	$168mW$
Wait for data	$9.6ms$	$254mV$	$50.8mA$	$168mW$
Send ACK	$280\mu s$	$285mV$	$57mA$	$168mW$
Wait to send	$1.57s$	$253.3mV$	$50.7mA$	$167mW$
Transmit data	$300\mu s$	$285mV$	$57mA$	$188mW$
Wait for ACK	$3ms$	$254mV$	$50.8mA$	$168mW$
Wake ping transmit	$280\mu s$	$285mV$	$57mA$	$168mW$
Wait for data	$9.6ms$	$254mV$	$50.8mA$	$168mW$

Based on the measurements above, a graph is plotted as seen in figure 4.24 in order to present the average power consumption at different wake ping intervals. In order to present the worst case values, *Wait to send* is set to one wake ping period. The graph is based on data transmission for every $150th$ wake ping interval, and data relaying every $100th$. This results in a 5 minute data transmission interval at the tested $2s$ wake ping interval. As can be seen, the average power of the ATmega328p with the custom multi-hop protocol at the $2s$ interval used for testing purposes is $5.34mW$.

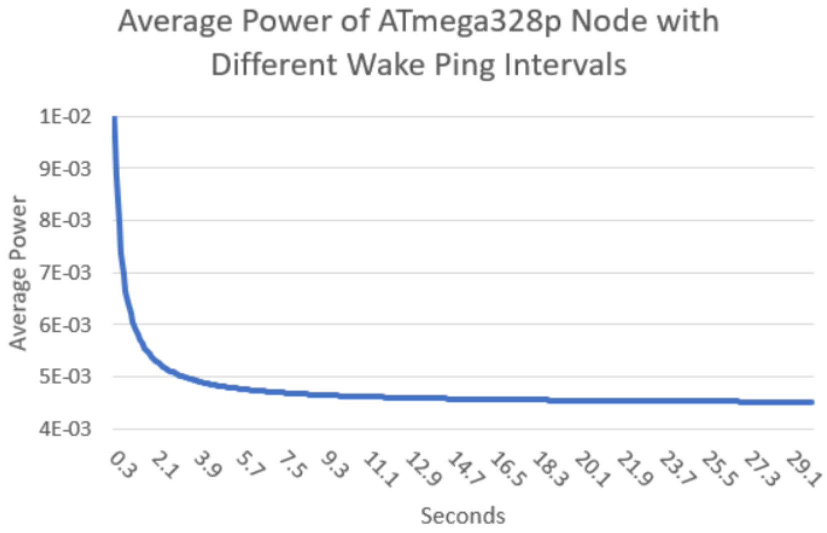


Figure 4.24: Average power consumption with different wake ping intervals with the ATmega328p node

When using a constant 5 minute interval for data transmission, and assuming that 3 relays are performed for each transmission, the optimal average power of $2.78mW$ was found at a $1.2s$ wake ping interval.

4.2.3 nRF24l01+PA+LNA

4.2.3.1 Range

The range of the nRF24l01+PA+LNA transceiver is tested by the use of a drone. The test is conducted in the municipality of Grimstad. Therefore, interference is to be expected, leading to a slightly shortened range. It is also unknown to what degree the radio signals from drone and controller have any impact on achieved range. A gateway is placed at the launch location of the drone, and a node is placed on the drone as shown in figure 4.25.

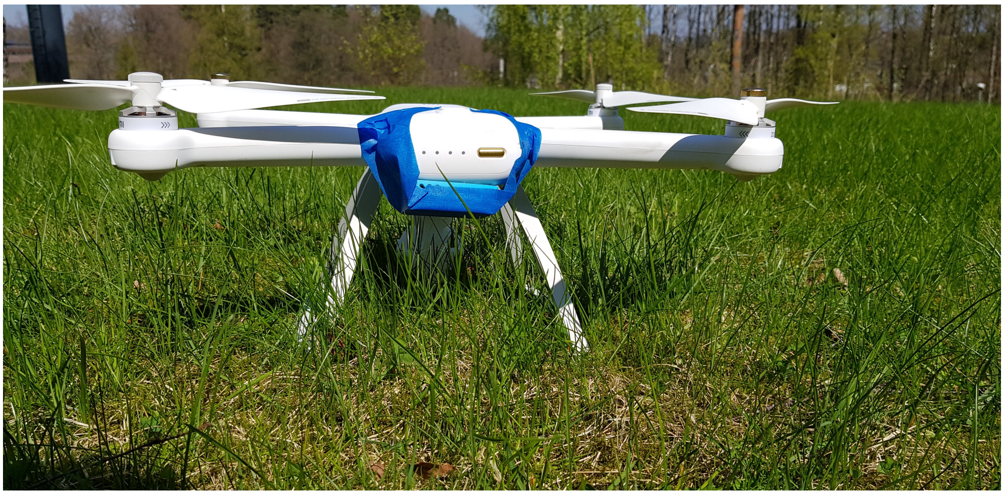


Figure 4.25: Drone with a sensor node placed on the bottom side

Figure 4.26 shows the approximate locations and result of range test for the ATmega328p node with transceiver nRF24l01+PA+LNA on both ends with *PALevel* set to *LOW*. At 228.5m distance 40m above ground level, approximately 50% of transmitted packages were lost. Test was conducted near University of Agder campus Grimstad.

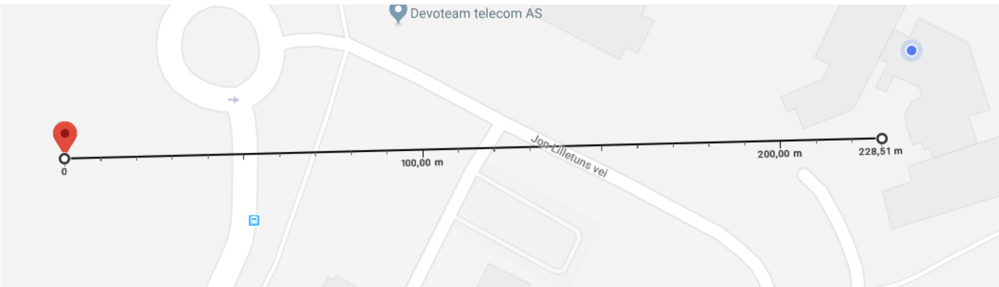


Figure 4.26: Map of test location and result

4.3 nRF52840 Node Test

It was discovered early in the project that 1.8V operation is far superior to 3.3V. The current consumption of 3.3V is higher than that of 1.8V. Therefore, 3.3V was discarded completely, and all nodes were configured with 1.9V. A test showing the energy consumption at 3.3V is presented in section 4.3.1.4

4.3.1 Energy Consumption

The energy consumption tests of the nRF52840 are conducted in the same manner as described in section 4.2.1 with the Keysight InfiniiVision DSOX2002A oscilloscope and a 5Ω resistor. All transmission tests are conducted with 1.9V supply and 8dBm coded transmissions unless other values are specified.

4.3.1.1 Sleep Consumption

Figure 4.27 shows the sleep current of the nRF52840 node with BME680 measured with the Keysight B2950A electrometer. The measurement was conducted over a 20s with 1.9V supply, during which the average current was 6.69μA, and the average power 12.7μW.

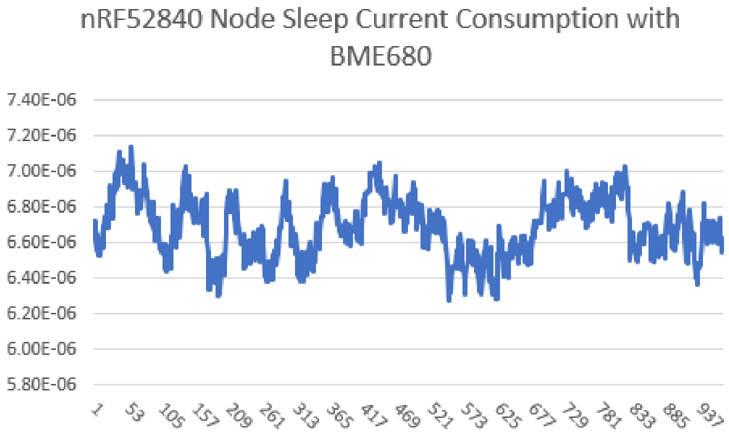


Figure 4.27: nRF52840 sleep current consumption

Figure 4.28 shows the sleep current of the nRF52840 node with HDC2010 with 1.9V supply. The average current was 3.65μA, and the average power 6.94μW.

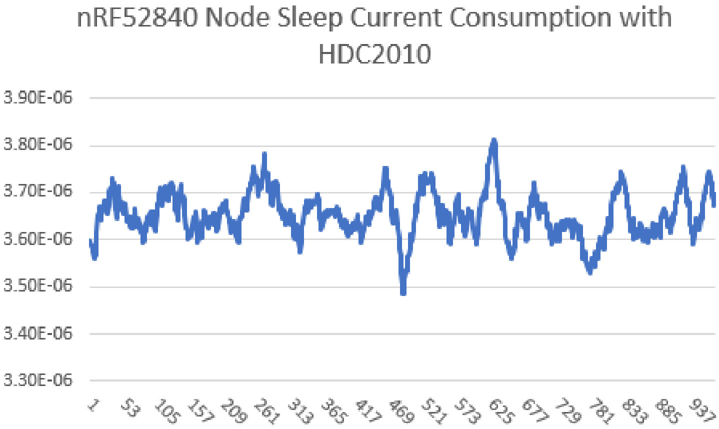
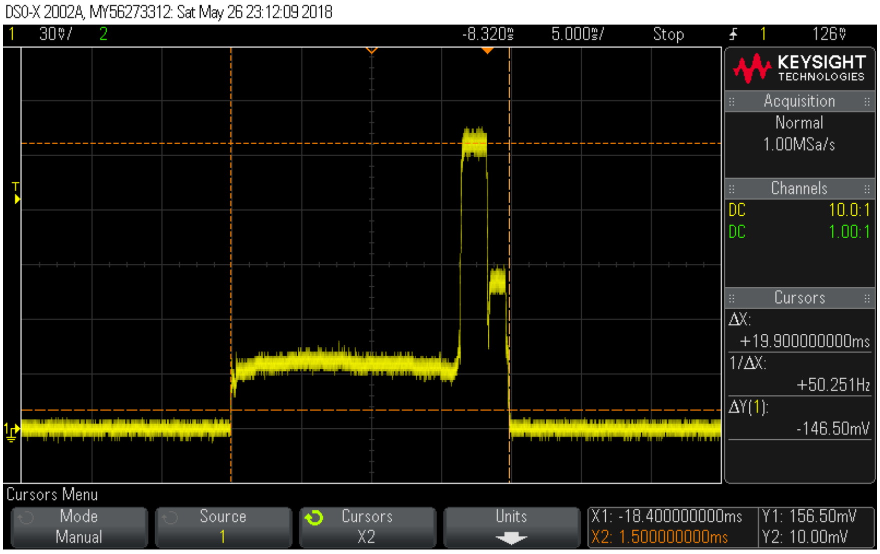


Figure 4.28: nRF52840 sleep current consumption with HDC2010 and 1.9V operation

4.3.1.2 Star

Figure 4.29 shows the measurement of the full on-time of one transmission from the nRF52840 in star topology. Cursors X1 and X2 indicate the total on-time of the node, while Y1 and Y2 indicates the peak voltage measured over the 5Ω resistor. As seen in the figure, the total on-time of the node is 19.9ms, which consists of a 16.5ms sensor read, 2ms transmission and 1.4ms ACK receival period.



DSO-X 2002A, MY56273312: Sat May 26 23:12:09 2018

1 30mV/ 2 -8.320ms 5.000ms/ Stop 1 126mV

Acquisition
Normal
1.00MSa/s

Channels
DC 10.0:1
DC 1.00:1

Cursors
ΔX:
+19.900000000ms
1/ΔX:
+50.251Hz
ΔY(1):
-146.50mV

Cursors Menu
Mode: Manual
Source: 1
Cursors: X2
Units

X1: -18.400000000ms
X2: 1.500000000ms
Y1: 156.50mV
Y2: 10.00mV

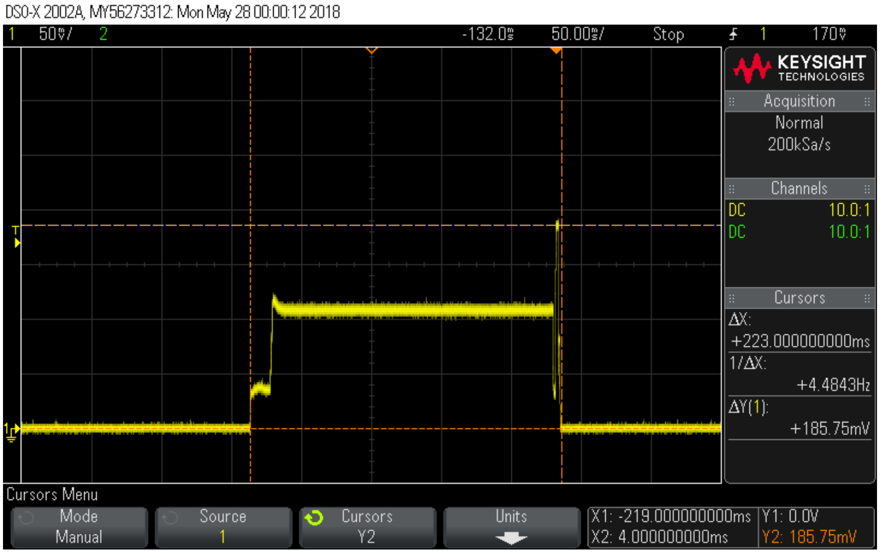
Figure 4.29: nRF52840 oscilloscope measurement in star topology with +8dBm transmission power and BME680

Table 4.5 shows the current and power consumption based on figure 4.29, where power is calculated with the assumption of 1.9V operation

Table 4.5: nRF52840 consumption with +8dBm and BME680 in star topology

	Period	**Measured Voltage**	**Current**	**Power**
Sensor read	$16.5ms$	$34.5mV$	$6.9mA$	$13.1mW$
Transmission	$2ms$	$156mV$	$31.2mA$	$59.3mW$
ACK	$1.4ms$	$79.5mV$	$15.9mA$	$30.2mW$

Figure 4.30 depicts the measurement of the 10th interval, where the BME680 heater is enabled for $200ms$ in order to get accurate IAQ readings. The current measured was $15mA$, and the power with $1.9V$ was $28.5mW$.



DSO-X 2002A, MY56273312: Mon May 28 00:00:12 2018

Acquisition: Normal, 200kSa/s

Channels: DC 10.0:1, DC 10.0:1

Cursors: ΔX: +223.000000000ms; 1/ΔX: +4.4843Hz; ΔY(1): +185.75mV

Cursors Menu: Mode Manual; Source 1; Cursors Y2

X1: -219.000000000ms; X2: 4.000000000ms; Y1: 0.0V; Y2: 185.75mV

Figure 4.30: nRF52840 oscilloscope measurement of the BME680 heater

Equation 4.6 presented in section 4.2.1 is used to calculate the average power of the node while active, where:

- P_{active} is the average power during on-state of the node
- P_{sensor} is the power consumed during sensor reading, here 13.1
- T_{sensor} is the period for sensor readings, here $16.5ms$
- P_{tx} is the power consumed during transmission, here $59.3mW$
- T_{tx} is the period for transmission, here $2ms$
- P_{ACK} is the power consumed during ACK receival, here $30.2mW$
- T_{ACK} is the period for ACK receival, here $1.4ms$
- P_{BME680_heater} is the power consumed by the BME680 heater, here $28.5mW$
- T_{BME680_heater} is the period for the BME680 heater. The period is $200ms$, however as this only occurs on every 10th transmission interval, the value used for calculating

P_{active} is $200ms/10 = 20ms$.

By inserting the measured values into equation 4.6, the average power while active is found in equation 4.9.

$$P_{active} = \frac{13.1mW \cdot 16.5ms + 59.3mW \cdot 2ms + 30.2mW \cdot 1.4ms + 28.5mW \cdot 20ms}{39.9ms} \\ = 23.7mW \tag{4.9}$$

By assuming a perfect transmission where the message arrives on the first attempt, and no retransmission is required, the average power consumption with different transmission intervals is found based on figures 4.27, 4.29 and 4.30 by using equation 4.10

$$P_{average} = \frac{23.7mW \cdot 39.9ms + 12.7\mu W \cdot T_{sleep}}{39.9ms + T_{sleep}} \tag{4.10}$$

By ranging T_{sleep} from $10s$ to $300s$, a graph is plotted as shown in figure 4.31. At $55s$ interval, the power consumption is $29.9\mu W$, while it goes below $20\mu W$ at $130s$. The graph presented is based on $+8dBm$ transmission power, as well as the use of BME680 while reading temperature, RH and pressure on every interval, and IAQ on every 10th interval.

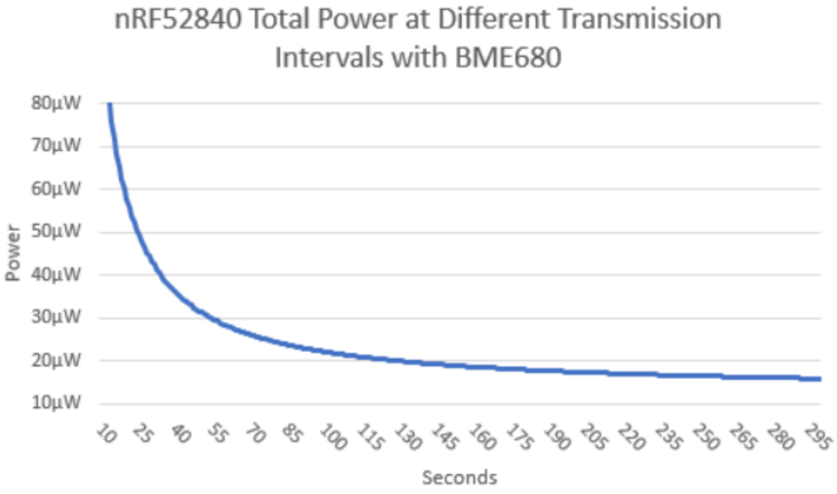
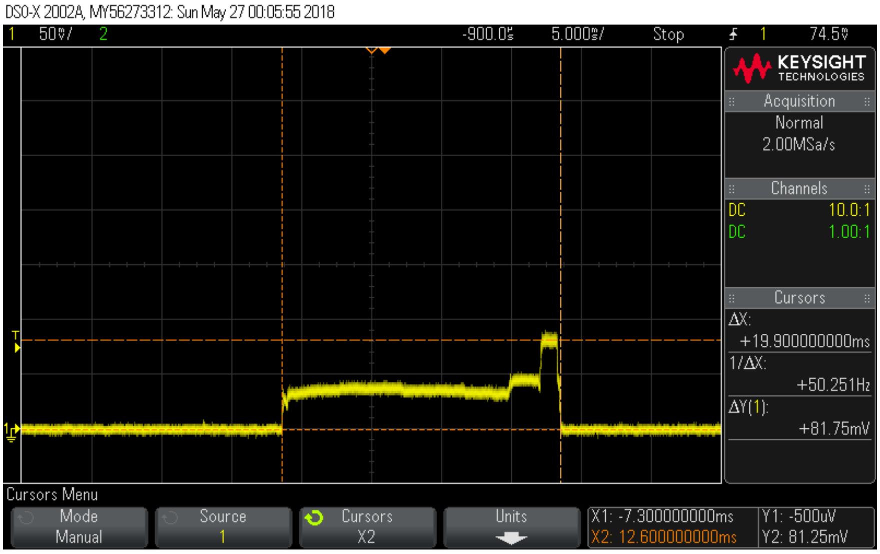


Figure 4.31: Average power consumption of nRF52840 in star topology with different transmission intervals

Figure 4.32 depicts the measurements with $-40dBm$ transmission power. The measured power consumption during transmission is $18.2mW$ while the average power with $55s$ transmission interval is $28.4\mu W$


DSO-X 2002A, MY56273312: Sun May 27 00:05:55 2018

1 50mV/ 2 -900.0µs 5.000ms/ Stop 1 74.5mV

Acquisition
Normal
2.00MSa/s

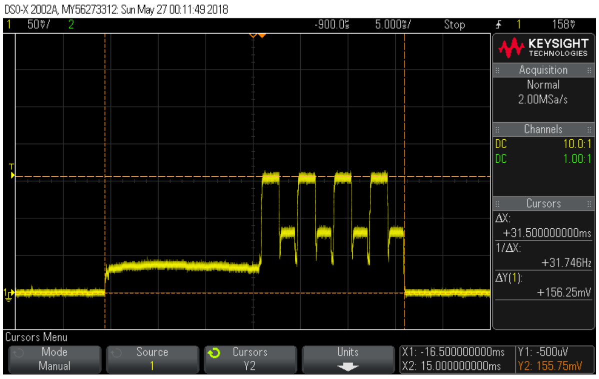
Channels
DC 10.0:1
DC 1.00:1

Cursors
ΔX:
+19.900000000ms
1/ΔX:
+50.251Hz
ΔY(1):
+81.75mV

Cursors Menu
Mode Manual
Source 1
Cursors X2
Units
X1: -7.300000000ms
X2: 12.600000000ms
Y1: -500uV
Y2: 81.25mV

Figure 4.32: nRF52840 measurement in star topology with -40dBm transmission power and BME680

In the case of a failed transmission, the node attempts to transmit its data three more times. This process is depicted in figure 4.33, where a transmission power level of $8dBm$ is used, as well as the BME680. Figure 4.34 presents a graph of the average power consumption at different transmission intervals in the worst case possible, with transmission power level of $8dBm$ and four failed transmissions on every transmission interval. The average power at $55s$ is $38.7\mu W$.


DSO-X 2002A, MY56273312: Sun May 27 00:11:49 2018

1 50mV/ 2 -900.0µs 5.000ms/ Stop 1 158mV

Acquisition
Normal
2.00MSa/s

Channels
DC 10.0:1
DC 1.00:1

Cursors
ΔX:
+31.500000000ms
1/ΔX:
+31.746Hz
ΔY(1):
+156.25mV

Cursors Menu
Mode Manual
Source 1
Cursors Y2
Units
X1: -16.500000000ms
X2: 15.000000000ms
Y1: -500uV
Y2: 155.75mV

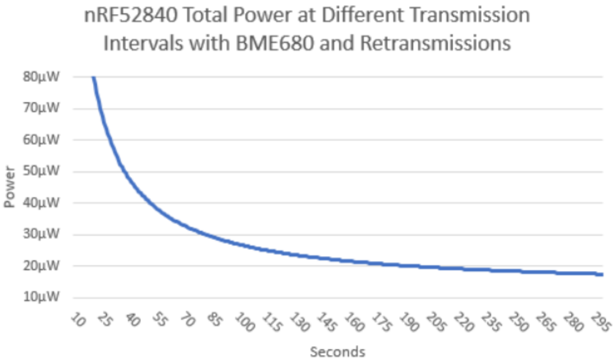
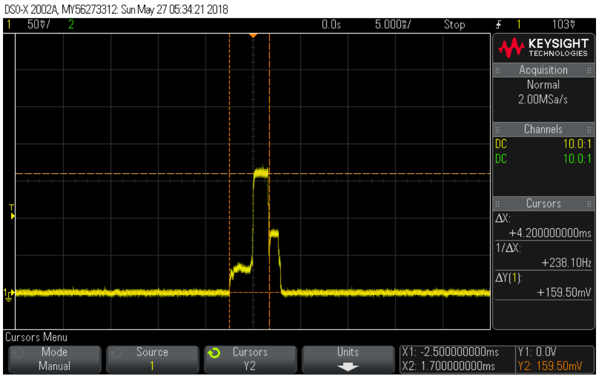


Figure 4.33: nRF52840 measurement in star topology with 8dBm retransmissions

Figure 4.34: Average power consumption of nRF52840 with 8dBm transmission power and failed transmissions

Figure 4.35 depicts the oscilloscope measurement of the nRF52840 node with $8dBm$ and the HDC2010 sensor. The transmission and ACK characteristics remain the same, however P_{sensor} and T_{sensor} are measured at $11.2mW$ and $2.3ms$ respectively. Additionally, as shown in figure 4.28, P_{sleep} is $6.94\mu W$. Considering these values, the graph in figure 4.36 is plotted. The average power consumption at $55s$ is $10.3\mu W$.



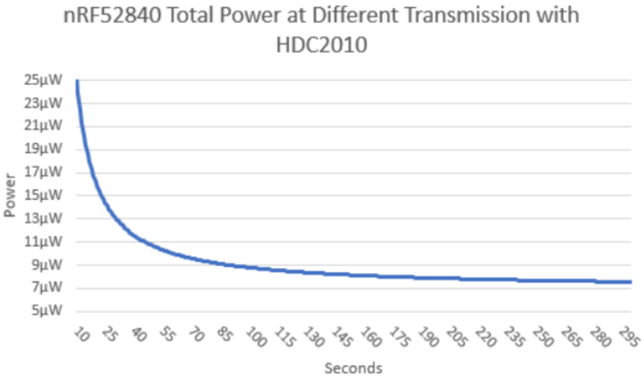
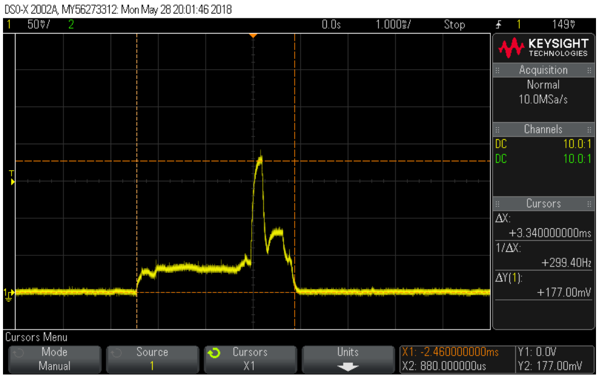


Figure 4.35: nRF52840 measurement in star topology with 8dBm and HDC2010

Figure 4.36: Average power consumption of nRF52840 with 8dBm transmission power and HDC2010

A test was conducted in order to establish the practical differences between coded and non-coded transmissions where a transmission rate of $1Mb/s$ was used. Figure 4.37 shows the oscilloscope measurements of the test, where the transmission time was $280\mu s$ and the ACK time $580\mu s$. Figure 4.38 shows the graph of average power consumption of the node at different transmission intervals, where a power of $8.1\mu W$ was obtained at a $55s$ interval.



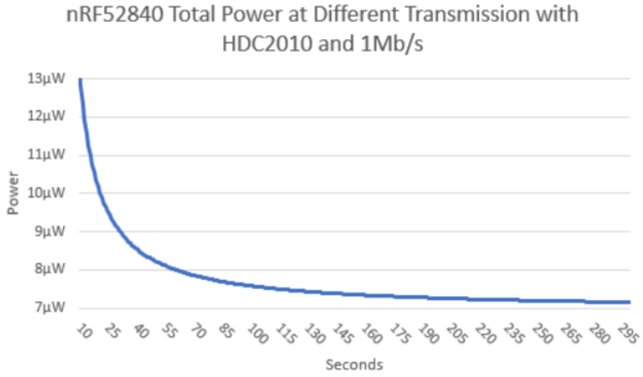
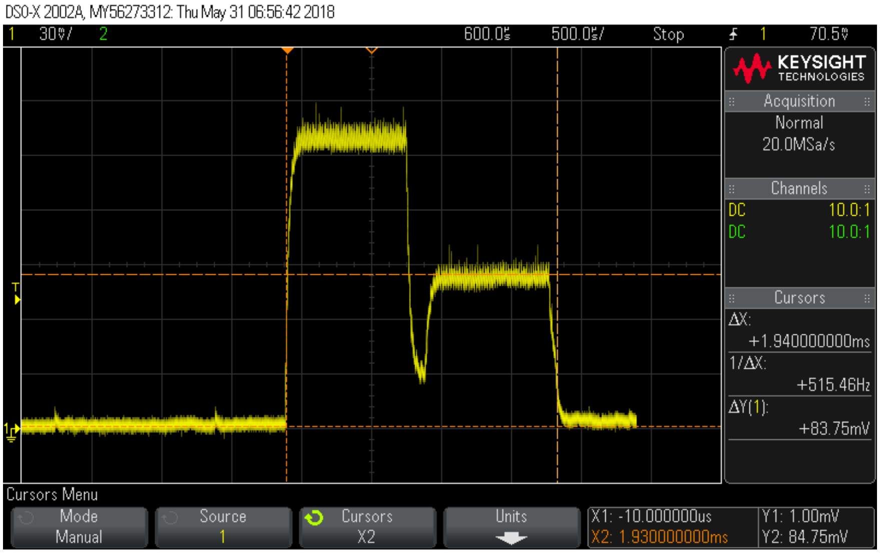


Figure 4.37: nRF52840 measurement in star topology with 8dBm, HDC2010 and $1Mb/s$ non-coded transmissions

Figure 4.38: Average power consumption of nRF52840 with 8dBm, HDC2010 and $1Mb/s$ non-coded transmissions

4.3.1.3 Multi-Hop

Figure 4.39 shows a single wake ping measured by an oscilloscope with a total period of $1.93ms$. All tests are conducted with BME680, $1.9V$ supply and $8dBm$ transmission power, with the exception of wake pings which are transmitted at one power level lower. Additionally, a wake ping interval of $2s$ is used.



DSO-X 2002A, MY56273312: Thu May 31 06:56:42 2018

Acquisition: Normal, 20.0MSa/s

Cursors: ΔX: +1.940000000ms; 1/ΔX: +515.46Hz; ΔY(1): +83.75mV

X1: -10.000000us; X2: 1.930000000ms; Y1: 1.00mV; Y2: 84.75mV

Figure 4.39: Oscilloscope measurement of nRF52840 multi-hop wake ping

Table 4.6 outlines the energy consuming elements of a single wake ping.

Table 4.6: nRF52840 wake ping measurements

	Period	**Measured Voltage**	**Current**	**Power**
MCU on	$220\mu s$	$34.5mV$	$6.9mA$	$13.1mW$
Wake ping transmit	$860\mu s$	$159mV$	$31.8mA$	$60.3mW$
Wait for data	$850\mu s$	$83.8mV$	$16.8mA$	$31.8mW$

Figure 4.40 shows the energy consumption while transmitting data, where the total measured period was $32.2ms$

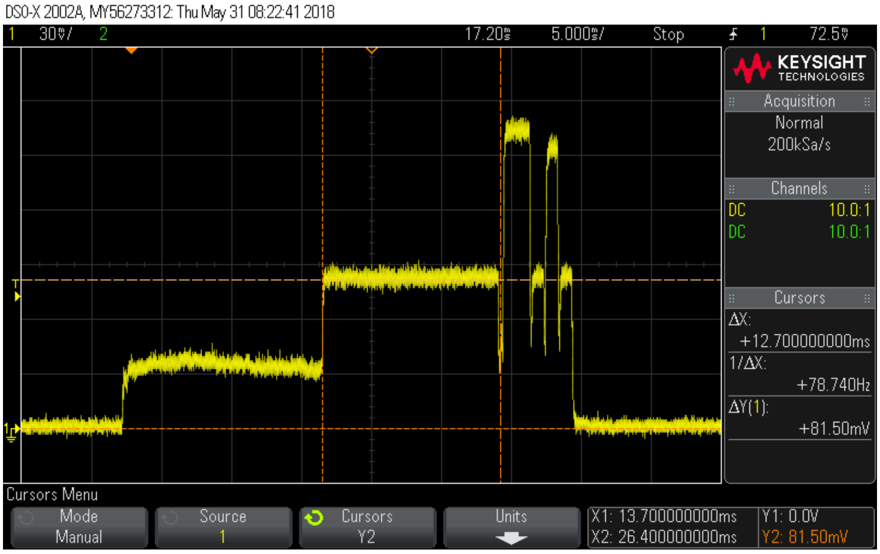


Figure 4.40: Oscilloscope measurement of nRF52840 multi-hop data transmission

Table 4.7 outlines the energy consuming elements of the measured transmission period. Carrier sense and random backoff were not successfully measured, however their maximum periods are known and the power consumption based on the peak of the measurement within their respective time period.

Table 4.7: nRF52840 data transmission measurements

	Period	**Measured Voltage**	**Current**	**Power**
MCU on	16.3ms	35mV	7mA	13.3mW
Wait to send	12.7ms	82.3mV	16.5mA	31.3mW
Carrier sense	128μs	83.5mV	16.7mA	31.7mW
Random backoff	80μs	83.5mV	16.7mA	31.7mW
Transmit data	1.9ms	163mV	32.6mA	61.9mW
Wait for ACK	900μs	82.3mV	16.5mA	31.3mW
Wake ping transmit	860μs	159mV	31.8mA	60.3mW
Wait for data	850μs	83.8mV	16.8mA	31.8mW

Figure 4.41 shows the energy consumption while transmitting data, where the total measured period was 2.03s. The figure presents a worst case scenario, where the *Wait to send* period was equal to a wake ping interval.

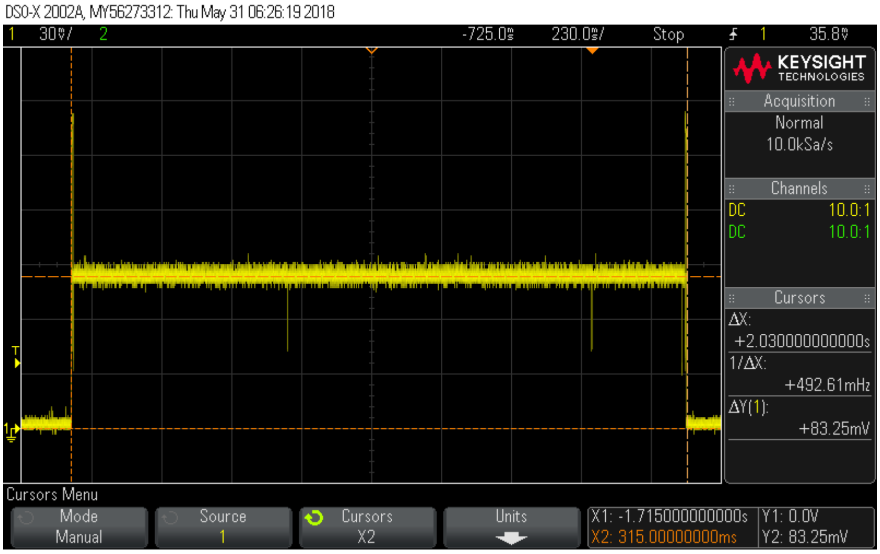


Figure 4.41: Oscilloscope measurement of nRF52840 multi-hop data relay

Figure 4.42 shows a zoomed image of before the "Wait to send" period in figure 4.41, while figure 4.43 shows a zoomed image of what happens after.

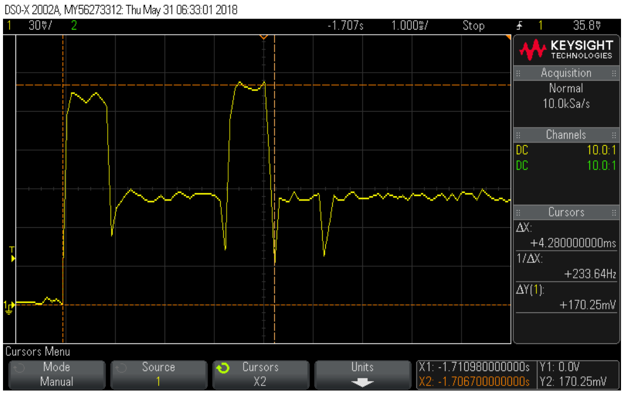


Figure 4.42: Figure 4.41 zoomed in before "Wait to send"

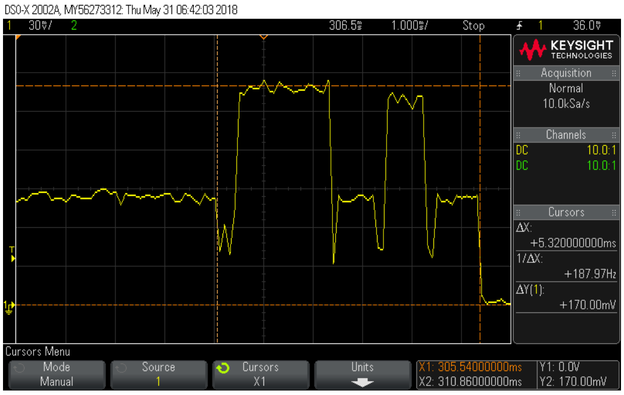


Figure 4.43: Figure 4.41 zoomed in after "Wait to send"

Table 4.8 outlines the energy consuming elements of the measured transmission period. As in table 4.7, the period and voltage measurement of carrier sense and random backoff are estimated.

Table 4.8: nRF52840 data transmission measurements

	Period	**Measured Voltage**	**Current**	**Power**
MCU on	$23.1ms$	$35mV$	$7mA$	$13.3mW$
Carrier sense	$128\mu s$	$83.5mV$	$16.7mA$	$31.7mW$
Random backoff	$80\mu s$	$83.5mV$	$16.7mA$	$31.7mW$
Wake ping transmit	$860\mu s$	$159mV$	$31.8mA$	$60.3mW$
Wait for data	$850\mu s$	$83.8mV$	$16.8mA$	$31.8mW$
Transmit ACK	$740\mu s$	$83.8mV$	$16.8mA$	$31.8mW$
Wait to send	$2s$	$82.3mV$	$16.5mA$	$31.3mW$
Transmit data	$300\mu s$	$163mV$	$32.6mA$	$61.9mW$
Wait for ACK	$900\mu s$	$82.3mV$	$16.5mA$	$31.3mW$
Wake ping transmit	$860\mu s$	$159mV$	$31.8mA$	$60.3mW$
Wait for data	$850\mu s$	$83.8mV$	$16.8mA$	$31.8mW$

Based on the measurements above, a graph is plotted as seen in figure 4.44 in order to present the average power consumption at different wake ping intervals. In order to present the worst case values, *Wait to send* is set to one wake ping period. The graph is based on data transmission for every $150th$ wake ping interval, and data relaying every $100th$. This results in a data transmission interval of 5 minutes at the tested $2s$ wake ping interval As can be seen, the average power of the nRF52840 with the custom multi-hop protocol at the $2s$ interval used for testing purposes is $892\mu W$.

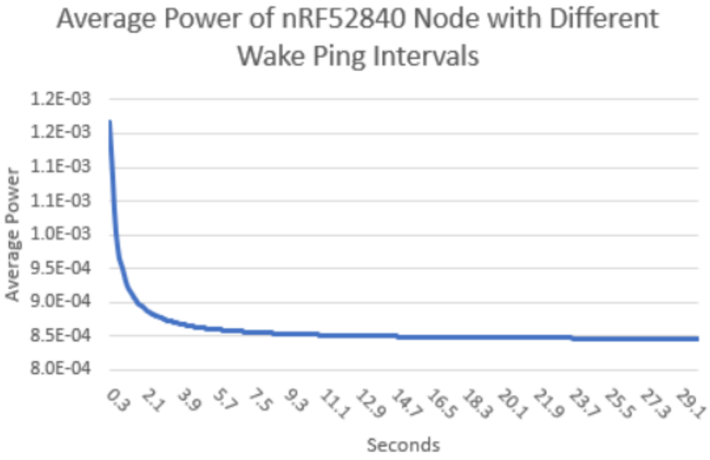
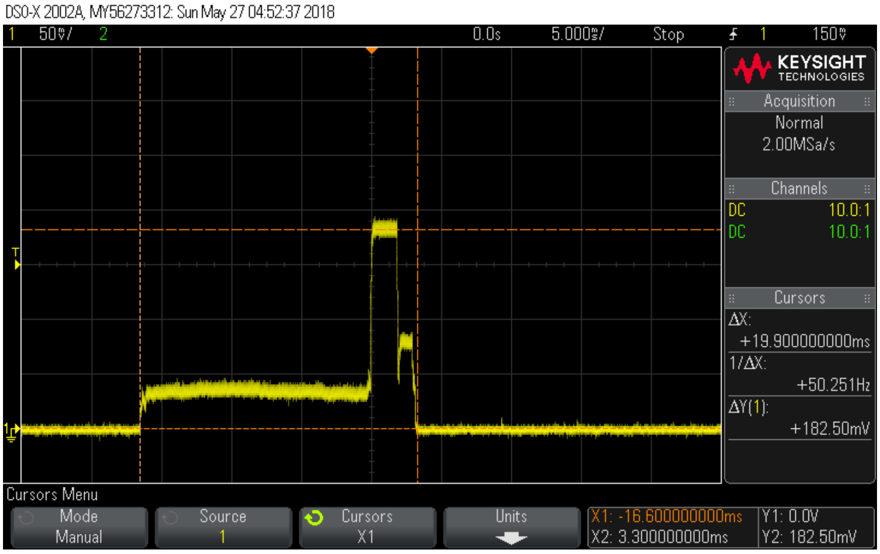


Figure 4.44: Average power consumption with different wake ping intervals with the nRF52840 node

When using a constant 5 minute interval for data transmission, and assuming that 3 relays are performed for each transmission, the optimal average power of $267\mu W$ was found at a $0.6s$ wake ping interval.

4.3.1.4 1.8V Operation

Measurements have been conducted with $3.3V$ operation as shown in figure 4.45. Based on the values presented in the figure, an average power with $55s$ transmission intervals of $48.2\mu W$ was calculated by using the method described in section 4.3.1.2. In comparison, the node with $1.9V$ supply, but otherwise identical configurations of transmission power and sensors, consumed $29.9\mu W$


DSO-X 2002A, MY56273312: Sun May 27 04:52:37 2018

1 50mV/ 2 0.0s 5.000ms/ Stop 1 150mV

Acquisition
Normal
2.00MSa/s

Channels
DC 10.0:1
DC 10.0:1

Cursors
ΔX:
+19.900000000ms
1/ΔX:
+50.251Hz
ΔY(1):
+182.50mV

Cursors Menu
Mode Manual
Source 1
Cursors X1
Units
X1: -16.600000000ms
X2: 3.300000000ms
Y1: 0.0V
Y2: 182.50mV

Figure 4.45: nRF52840 oscilloscope measurement in star topology with $8dBm$ transmission power, $3.3V$ supply and BME680

As all components are rated for voltage supply of less than $1.9V$, everything worked accordingly when changing the output voltage of the BQ25570 buck converter with the exception of the TPS22860 switch, which is used for battery level sensing, with the use of a supercapacitor. Whenever the supercapacitor was charged to approximately $5V$, the switch would not turn on. This is caused by the logic level voltage being dependent on V_{BIAS}, which is connected directly to the supercapacitor or battery. As the graph in figure 4.46 from the TPS22860 datasheet shows, the voltage required to turn on the switch with $V_{BIAS} = 5V$ is $V_{IH} \approx 2.1V$, while the actual ON-signal is $1.9V$.

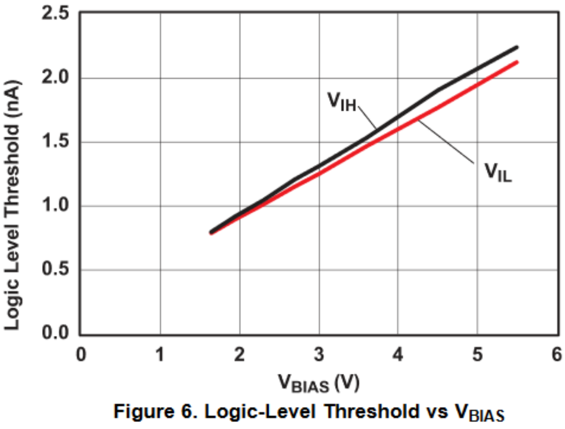


Figure 4.46: Logic-level threshold of TPS22860 [5]

4.3.1.5 Range

The nRF52840 range is tested in the same manner as the nRF24l01+PA+LNA as described in section 4.2.3 by the use of a drone. Achieved range with transmission power set to $8dBm$ and bitrate to $125kb/s$ coded with line of sight was approximately $1.8km$. The test was conducted near University of Agder campus Grimstad.

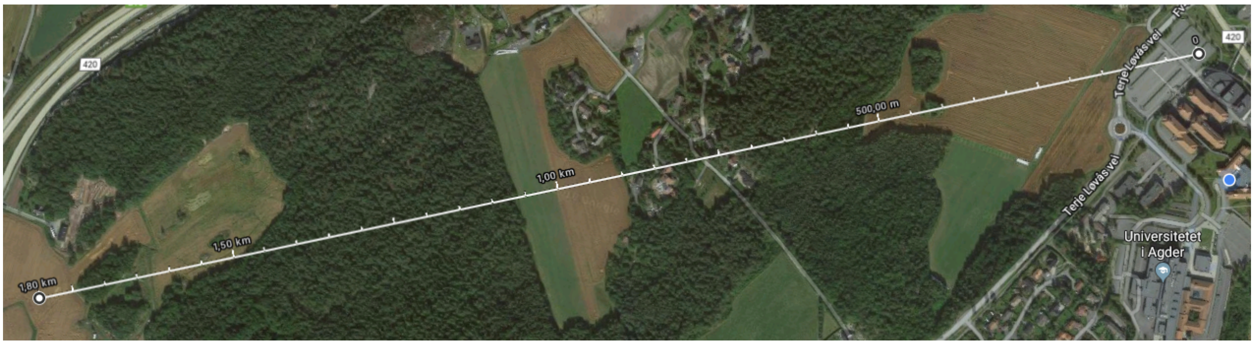


Figure 4.47: nRF52840 range test with $8dBm$ transmission power and coded transmissions

Figure 4.48 depicts the range test of the nRF52840 node with $8dBm$ transmission power and non-coded $1Mb/s$ transmissions. The range obtained during testing was approximately $775m$.

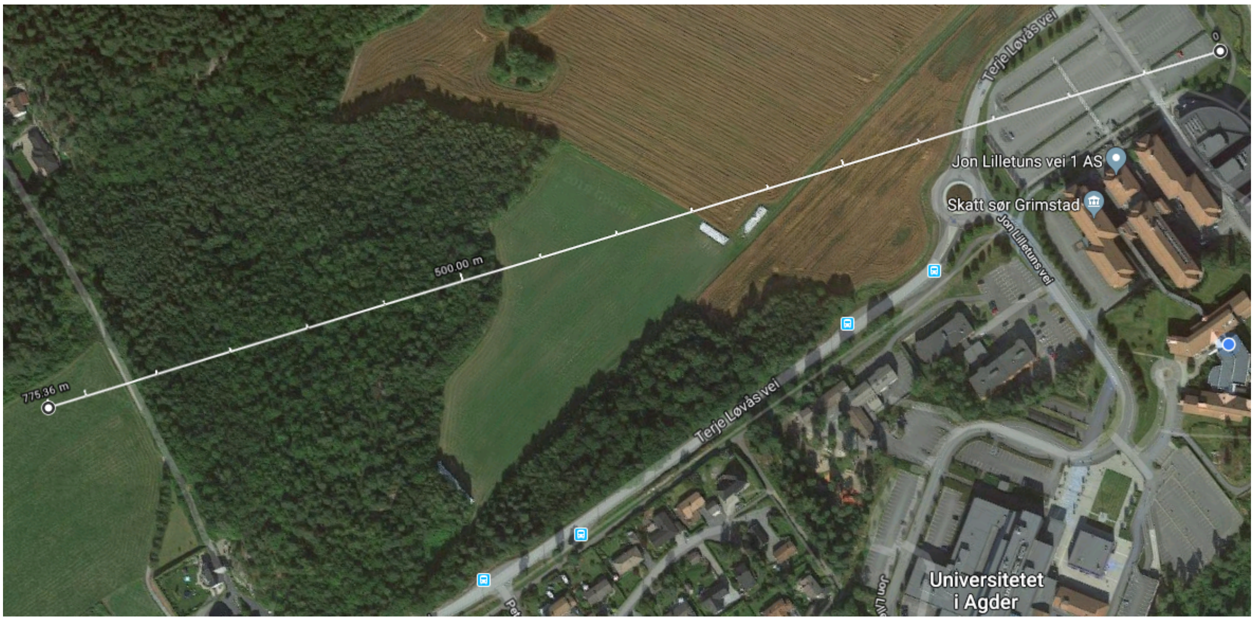


Figure 4.48: nRF52840 range test with $8dBm$ transmission power and non-coded transmissions

Chapter 5

Discussion

5.1 MG1 - Develop ATmega328p WSN

MG1 in section 1.3.1 states that a WSN should be developed with a design based on the ATmega328p MCU along with the nRF24l01+ for wireless communication. Firmware should be programmed, and a PCB designed.

5.1.1 Hardware

A PCB has been designed as presented in section 3.4.1.2. The design includes an EH circuit, the MCU, the nRF24l01+PA+LNA transceiver and the necessary sensors. The schematics and PCB design are found in appendix D.1.

The PCB has support for all the sensors used throughout the project, namely MAX44009, HDC2010, BME680 as well as the battery level sensing circuit. The voltage supply from the BQ25570 buck converter is $3.3V$, and the BQ25570 is programmed to ensure that the buck converter is disabled before the output voltage can fall below $3.3V$ as a cause of low battery voltage. By doing this, the BQ25570 is able to recharge the battery to a programmed value hysteresis value above the cutoff. During testing, the PCB was found to work as expected, with no detectable errors or faults.

The nRF24l01+PA+LNA modules used for this project did however prove unstable as explained in section 5.6.

5.1.2 Firmware

Developing code for the ATmega328p node proved, as expected, to be an uncomplicated task. Libraries, good documentation and easy to use Arduino functions helps expedite the developing process. Most of the components used for the nodes had pre-written libraries by the Arduino community with the exception of HDC2010 and MAX44009 as these are recently released on the consumer market. Therefore, the code for these had to be developed by using the values and registers specified by their associated datasheets.

In order to allow for a single firmware to support the various sensors used for the nodes, an I^{2}C scanner is implemented. The scanner allows the node to identify which sensors are attached on startup from a list of known I^{2}C slave addresses. The connected sensors are enabled for operation, while those not connected are never attempted to be read by the MCU.

Low power mode is attained by using the ATmega328ps watchdog timer. This allows the MCU to sleep while the $128kHz$ crystal runs, creating a wake interrupt after a maximum continuous period of $8s$. This duration can be looped to extend the sleep duration.

Wireless communication was achieved using the nRF24l01+PA+LNA radio module. With the help of libraries and simple operation, this module is an easy introduction to wireless communication. However, when trying to optimize there some functionalities missing such as RSSI and coded transmissions that would improve the WSN considerably.

5.2 MG2 - Develop nRF52840 WSN

MG2 in section 1.3.2 states that a WSN should be developed with a design based on the nRF52840 MCU. Firmware should be programmed, and a PCB designed. The nRF52840 IC may however not be on the market during the course of the project, so the PDK may be used instead.

5.2.1 Hardware

A PCB has been designed based on the BMD-340 module containing the nRF52840 IC as well as an on-board antenna. This is, however, not released during the course of the project, and has therefore not been produced or tested. The PDK is therefore used in order to develop a proof-of-concept nRF52840 node, and a breakout board containing the EH circuit as well as all the sensors has been designed and used during testing. The circuit and PCB is described in section 3.6.1, and the full schematics and PCB design can be found in appendix D.2 and D.3.

Two different generations of nRF52840 PDK have been used for the project. During testing, it was discovered that the older generation consumed an added constant $2mA$ of current.

The same behaviour was discovered when powering the on-board Segger IC used to program the PDK via USB, leading to the assumption that the Segger IC was always powered on despite turning its voltage supply off with the on-board switch. Therefore, the old generation was not used for energy consumption tests. Additionally, the two different generations have radio and SDK compatibility issues that were not discovered until late in the project, resulting in misspent time. In the initial phase of the project, nRF52840 PDK v0.9.2 was used for testing, while using SDK v14.2.0. Additional nRF52840 PDKs with the version number 0.11.0 were obtained during the course of the project. These worked successfully for all operations in the star topology firmware. When using the nRF52840 v0.11.0 PDKs for the multi-hop system, however, they were not able to communicate with each other, using the exact same code as with the nRF52840 v0.9.2 PDKs. It was later identified that SDK 15.0.0 had been released, which supports nRF52840 v0.11.0, after the new PDKs were obtained, however the firmware was finished and testing was initiated, leaving no time to rewrite the code for the new SDK. Multi-hop was nonetheless tested successfully with the nRF52840 v0.9.2 PDKs.

As the nRF52840 gateway designed for the project, which is described in section 3.8, contains two separate radios, unwanted interference may occur between them. Appendix D.4 presents the full schematics and PCB design. The gateway has not been built and tested due to the nRF52840 IC not being released at the time of the project. Therefore, no tests have been conducted in order to establish whether or not there is interference between the antennas. If, however, interference should prove to be a problem, the following measures may be taken to reduce or avoid it:

- Enlarge the PCB design in order to move the antennas further apart
- Install an interference shield between the two radios
- Separate ground planes with filter between them
- Discard Wi-Fi completely in favor of ethernet

5.2.2 Firmware

Bluetooth 5 is not used for the nodes as it leads to more overhead in the transmissions. Additionally, no connections are to be made with other Bluetooth enabled devices, making the protocol superfluous for use with the nodes. Therefore, a custom protocol is designed, which utilizes features provided by the Bluetooth 5 stack, such as coded transmissions.

In order to obtain the highest possible ranges of the nRF52840 nodes, coded transmissions are implemented. By doing this instead of using regular transmissions, the theoretical maximum range of the nodes in line of sight is increased from $200m$ to $1.6km$, as presented in section 2.3.1. The tests showed, however, that the actual obtainable ranges in line of sight were much higher, and by using a drone, ranges of $775m$ and $1.8km$ were measured. With a transmission time of $2ms$ and average power consumptions as low as $10.3\mu W$ at $55s$ transmission intervals when using coded and $8.1\mu W$ with non-coded $1Mb/s$ transmissions, the additional power consumption when using coded is not considered an issue when considering the range gain.

Developing with the Nordic Semiconductors SDK environment proved to have a steep learning curve since no member of the group had any prior experience with the platform. This combined with unfamiliar coding methods made the process time consuming. During the project however, the environment became more familiar, and developing new functionality became less time consuming. Developing I^{2}C code for the sensors proved to be one of the major tasks, particularly for the BME680 which required multiple different I^{2}C interactions. No libraries with the TWI structure were available for any of the sensors implemented in the node, and the TWI example provided with the SDK gave limited insight of how to design custom I^{2}C code. Code for all the sensors was however successfully developed, giving accurate control over the functionality of each sensor.

Low power mode on the nRF52840 node was achieved by using the application timers, and utilizing the low power LFCLK whenever no events are running. Application timers running on the LFCLK will then generate an interrupt, and perform the desired event.

5.3 MG3 - Energy Harvesting

MG3 in section 1.3.3 states that an EH circuit should be designed for the use of a PV panel. The EH circuit should include a battery, battery charger and step down converter to obtain the correct voltage for the nodes.

The EH circuit presented in section 3.1 proved functional for indoor and outdoor use. The circuit has been tested with both a $0.15W$ polycrystalline and a $0.36W$ monocrystalline PV panels, as well as both a $120mAh$ Li-Ion battery and a $5F$, $6V$ supercapacitor.

5.3.1 PV Harvesting

Preliminary testing with a monocrystalline PV panel with a size of $63x63mm$, rated to $0.36W$ max, produced power in abundance. A decision was therefore made to implement a smaller panel in the design, namely a $53x30mm$ panel rated to $0.15W$ advertised as monocrystalline. Upon the arrival and inspection of the small PV-panels, it was however discovered that they were composed of polycrystalline cells. At first, this was not regarded as a big problem, as most sources stated that there were only a few percentages of difference in efficiency between mono- and polycrystalline cells. The characterization of the polycrystalline panel also yielded results comparable to that of the specification. After conducting a full scale test of the energy harvesting and storage system, however, it was found that the panels proved insufficient. Finding reliable sources proved difficult, as most of the research in the field of photovoltaics are focused on outdoor applications in direct sunlight or clouded skies. Test results described in article [6] however, states that the low light efficiency of mono- and polycrystalline cells produced with conventional methods might differ with as much as 500% in efficiency in common indoor lighting, making it reasonable to conclude that migrating to a monocrystalline panel is required to have a fully functional system for indoor usage. Figure 5.1 depicts the difference in efficiency in light levels commonly found indoors with different PV technologies.

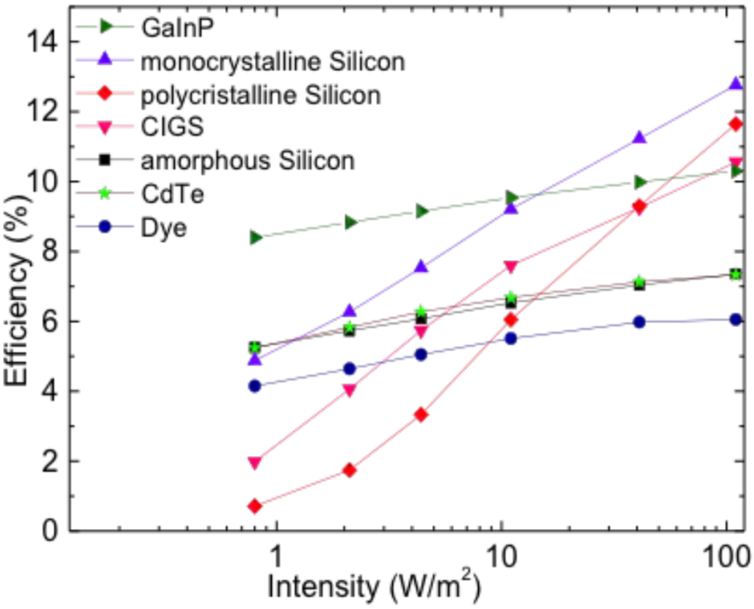


Figure 5.1: PV efficiency in low light conditions [6]

As described in section 4.1, the difference between monocrystalline and polycrystalline PV panels is substantial when tested under similar conditions indoors in a bright room. While testing with the battery, the polycrystalline panel provided $213.96\mu W$ average, while the monocrystalline panel provided $941.94\mu W$. Taking into account the difference in power ratings of the PV panels, the polycrystalline panel has an efficiency of 54.5% relative to the monocrystalline panel. As for supercapacitor charging, the relative efficiency is 58.5%. It can therefore be concluded, by using the worst-case results, that the polycrystalline panel used for this project has approximately 54.5% the efficiency of the monocrystalline panel while used indoors in a well-lit room.

As for the low light test, the Mooshimeter was unable to correctly measure the low current levels from the polycrystalline panel. A voltage rise on the battery of $0.005V$ was however observed. The monocrystalline panel provided an average of $212\mu W$, and the test showed a total battery voltage increase of $0.047V$. An approximated relative efficiency can be calculated based on the voltages as the voltage range is sufficiently narrow, and hence the voltage curve versus charge current is close to linear. The efficiency of the polycrystalline panel is 25.5% relative to the monocrystalline panel in low light conditions.

The tests conducted with a $5F$ supercapacitor as storage medium proved partially successful. This is further discussed in section 5.8.

5.4 MG4 - Research Sensors

MG4 in section 1.3.4 states that ULP sensors should be researched in order to find the most suitable sensors for the nodes. The environmental values of interest are temperature, RH, visible light, atmospheric pressure and IAQ.

Several sensors have been researched throughout the project. The BME680 measures all the values of interest except for visible light in a low power manner, with the exception of IAQ which requires $13mA$ for extended periods in order to heat up to sufficient temperatures. It is, however, monetarily costly. Therefore, the most likely sensor to be implemented in most nodes is the HDC2010, which is offered at a much lower cost than and BME680. It has an extremely low energy consumption during both sleep and measurement periods, making it a viable option for low light environments. The datasheet specifies a consumption of $550nA$ with a measurement interval of $1s$ while measuring both temperature and RH. IAQ and atmospheric pressure are oftentimes superfluous in well ventilated office- or household environments.

For measuring visible light, the MAX44009 is selected, as it offers sufficient precision at a low cost, while consuming extremely low amounts of power.

5.5 MG5 - Wireless Network Topologies

MG5 in section 1.3.5 states that the WSNs should be configured and tested with both one-hop and multi-hop topologies.

Disregarding hardware issues with the nRF24l01+PA+LNA transceiver on the Atmega328p node, the one-hop star topology has worked well. Long wake-up intervals, few wake-ups and short transmissions have ensured low power consumption. Collision mitigation mechanisms such as carrier sensing have proven to be somewhat superfluous for star topologies, as the probability of two or more nodes trying to transmit at the same time is very small. Using transmission time for the nRF52840 node with coded $125kb/s$ including waiting for and receiving ACK, one node with a transmission interval of $55s$ take up only $\frac{3.4ms}{55003.4ms} \approx 0.006‰$ of on-air time. This functionality may however prove more useful in future deployments with higher network density, shorter intervals and larger payloads. Dynamic transmission power adjustment has worked well with the nRF52840 nodes, effectively reducing power consumption and interference caused by the nodes. As for the Atmega328p node, due to issues and limitations with the transceiver, adjustment of transmission power had little or no effect.

As for the multi-hop topology, a higher level of complexity proved to be a challenge, both in terms of implementation and testing. The dynamic nature of the self-designed protocol and the unpredictable range of transceivers indoors makes it difficult to predict where packages are routed. As for the functionality of the multi-hop protocol, the results have been good. Packages are successfully routed to the gateway and nodes can be introduced into the network and freely moved around while still finding routes to the gateway. Medium access mechanism with random backoff followed by channel sensing is more important for the multi-hop topology, as transmissions are initiated by an external event instead of just the internal clock as in the star protocol. There are however some aspects that can and should be optimized for future implementations. Frequent wake-ups to broadcast the *wake_ping* messages consume a high amount of power. This wake-up interval should be adaptive to each network deployment. Additionally, scaling the wake-up interval with energy storage levels may help mitigate issues with energy holes in the network, where the energy storage medium the nodes that forward a large number of messages are depleted, leaving a hole in the topology. Scaling the interval may however lead to less optimal routes or higher latency in the network. Less frequent *wake_ping* messages may also lead to longer wake periods when nodes attempt data transmission, which also consumes energy. The density of the network has a large impact on this waiting period, as there are more nodes that can forward messages within range as the networks get denser.

5.6 MG6 - Test and Compare the WSNs

MG6 in section 1.3.6 states that extensive testing should be conducted in order to compare the two WSNs with various sensors and topologies.

During testing of the ATmega328p node, it was discovered that the nRF24l01+PA+LNA transceiver used for the project was highly unstable, likely due to poorly matched antennas and RF filters. As a result of this, the transceivers were unable to transmit at power levels higher than LOW, and even then, many transmissions were lost. This is likely an issue with the specific module obtained for the project, and not all nRF24l01+PA+LNA modules. Nevertheless, tests with transmission power level of LOW were conducted and treated as if the modules were in perfect order as to present results with a functional module.

5.6.1 Star Topology

The average power of the nRF52840 node in star topology with BME680 and $8dBm$ transmission power at $55s$ was $29.9\mu W$, while the power at $-40dBm$ is $28.4\mu W$. The difference in the average power consumption with the maximum and minimum transmission power levels are therefore considered to be insignificant, which is due to the short transmission interval of the nodes. With HDC2010 and $8dBm$ transmission power on the other hand, the average power consumption was measured to be $10.3\mu W$ with a $55s$ transmission interval. As stated in section 2.1.2, the leakage of the LIR2450 battery is approximately $11.1\mu A$, or $11.1\mu A \cdot 3.6V = 40\mu W$ at its nominal voltage. Therefore, the power consumption with the use of HDC2010 is approximately 20% of the total consumption of the system.

The ATmega328p node on the other hand, which was tested with LOW transmission power level, measured an average power of $42.7\mu W$ at a $55s$ interval with BME680, while the average power with HDC2010 was $24.8\mu W$.

Table 5.1 presents a comparison of the battery lifetime with each node as well as both the BME680 and HDC2010. The values are calculated by using equation 5.1 with the assumption of a fully charged battery with $120mAh \cdot 3.6V = 432mWh$ of available energy.

$$T_{days} = \frac{P_{battery}}{P_{battery\_leakage} + P_{node\_consumption}} \cdot \frac{1}{24^{h}/_{day}} = \frac{432mWh}{40\mu Wh + P_{node\_consumption}} \cdot \frac{1}{24^{h}/_{day}} \tag{5.1}$$

Table 5.1: Comparison of battery lifetime from full charge in star with 55s transmission interval with no energy harvesting

	Node power	**Days**
nRF52840 8dBm BME680	29.9μW	257.5
nRF52840 8dBm HDC2010	10.3μW	357.9
ATmega328p LOW BME680	42.7μW	217.7
ATmega328p LOW HDC2010	24.8μW	277.8

Table 5.2 presents the same comparison as table 5.1, however with the use of a supercapacitor instead of a battery. The worst case self discharge rate of $36\mu A \cdot 5.28V = 190.1\mu W$ as described in section 2.1.2, as well as the capacities of 7.47mWh and 1.87mWh for 1.8V and 3.3V respectively presented in section 3.1.4, are used.

Table 5.2: Comparison of 5F supercapacitor lifetime from full charge in star with 55s transmission interval with no energy harvesting

	Power capacity	**Node power**	**Days**
nRF52840 8dBm BME680	7.47mWH	29.9μW	1.41
nRF52840 8dBm HDC2010	7.47mWH	10.3μW	1.55
ATmega328p LOW BME680	1.87mWH	42.7μW	0.33
ATmega328p LOW HDC2010	1.87mWH	24.8μW	0.36

5.6.2 Multi-Hop Topology

Both the ATmega328p and the nRF52840 node have been tested in multi-hop topology with success. A graph has been plotted in order to present the average worst case power consumption of each node at various wake ping intervals, where the nodes data is transmitted on every 150th wake ping interval, and an assumed interval for relaying data at every 100th wake ping interval. The BME680 sensor was used during testing.

The nRF52840 node proved successful during multi-hop testing, both for indoor and outdoor use, with some limitations. For indoor use, the location of the nodes should be sufficiently well-lit, such as near a window or an incandescent light, and the node should use a monocrystalline panel of sufficient power rating. The test showed an average worst case power consumption of 892μW with a 2s wake ping interval, which is lower than the average harvested energy of 1.25mW in a well-lit room. When considering a 5 minute data transmission interval, the optimal wake ping interval is $0.6s$, at which the node consumes an average worst case power of $267\mu W$, effectively meaning that the harvested energy is $1.25mW/267\mu W = 4.68$ times higher than the power consumption.

The test of the ATmega328p node proved partially successful, however it is not plausible for indoor use. With a $2s$ wake ping interval, the average worst case power consumption was measured to be $5.34mW$, which is much higher than the harvested energy indoors. For the ATmega328p node, the optimal wake ping interval with 5 minute transmission intervals was found to be $1.2s$, at which the node consumes an average of $2.76mW$. Compared to the nRF52840, the ATmega328p therefore consumes 10.4 times as much power.

Similar to the star topology comparison in section 5.6.1, two tables are created in order to determine the lifetime of the node with battery and supercapacitor where the same capacities and self discharge values are used. Table 5.3 presents the lifetime in days with the use of the $120mAh$ battery with a total power capacity of $432mWh$, while table 5.4 presents the lifetime in hours with the use of the $5F$ supercapacitor, which has a power capacity of $7.47mWh$ for $1.9V$ operation and $1.87mWh$ for $3.3V$. Both the $2s$ wake ping values and the optimal wake ping interval at 5 minute transmission intervals are presented, which is $0.6s$ for nRF52840 and $1.2s$ for ATmega328p.

Table 5.3: Comparison of battery lifetime from full charge in multi-hop with no energy harvesting

2s Wake ping	**Node power**	**Days**
nRF52840	$892\mu W$	37.6
ATmega328p	$5.34mW$	6.9
Optimal wake ping interval		
nRF52840	$267\mu W$	58.6
ATmega328p	$2.76mW$	6.4

Table 5.4: Comparison of $5F$ supercapacitor lifetime from full charge in multi-hop with no energy harvesting

2s Wake ping	**Power Capacity**	**Node power**	**Hours**
nRF52840	$7.47mWH$	$892\mu W$	6.9
ATmega328p	$1.87mWH$	$5.34mW$	0.34
Optimal wake ping interval			
nRF52840	$7.47mWH$	$267\mu W$	16.34
ATmega328p	$1.87mWH$	$2.76mW$	0.63

5.6.3 Range

As presented in section 4.26, the ATmega328p node achieved a line of sight range of $228.5m$ with LOW transmission power level. Meanwhile, the nRF52840 node achieved $1.8km$ with coded transmissions, and $775m$ with non-coded.

5.6.4 Reliability

Both the star and multi-hop topologies utilize ACK messages in order to ensure the reception of data. Therefore, the probability of a packet loss is both considered and observed as low.

5.7 SG1 - 1.8V Supply for nRF52840

As the nRF52840 supports $1.8V$ operation, effectively resulting in a reduced energy consumption, the peripheral components chosen throughout the project should also support $1.8V$ to avoid the need for two separate voltage converters, as stated by SG1 in section 1.3.7.

The BQ25570 was successfully programmed for $1.9V$ operation with both supercapacitor and battery as storage element, as the resistors for $1.8V$ were not available at the time of construction. The end result was partially successful, as the entire system worked as intended with the exception of the TPS22860 switch used for voltage level sensing when using a supercapacitor at high voltage, as described in section 4.3.1.4. A different switch with a more suitable logic level threshold needs to be implemented in the nodes with $1.9V$ supply and a supercapacitor as energy storage. Alternatively, a transistor or mosfet may be used.

Besides the TPS22860 issue, a lower energy consumption was obtained. At a $55s$ transmission interval, the node with $3.3V$ supply and BME680 consumed an average of $48.2\mu W$, while the node with $1.9V$ consumed $29.9\mu W$. As a result of these findings, the BQ25570 output voltage was programmed to $1.9V$ on all nRF52840 nodes.

5.8 SG2 - Supercapacitor

SG2 in section 1.3.8 states that a supercapacitor should be implemented and tested as an alternative to a battery.

During the brief testing of supercapacitors, the results were mixed. The supercapacitors have a much higher self discharge compared to Li-Ion batteries. Additionally, when completely depleted, the BQ25570 is unable to charge the supercapacitors with a charge current higher than the leakage current of the supercapacitors in an indoor environment. Therefore, the supercapacitors must be pre-charged via the USB charger on the nRF52840 node design, or via an external source such as a regular, variable power supply.

With that in mind, the supercapacitor design proved viable in areas with sufficient lighting. When the supercapacitor is pre-charged, the BQ25570 delivers sufficient charging current in normal, indoor lighting. The BQ25570 ensures that the buck converter powering the nodes is turned off when the supercapacitor voltage falls below a programmed value, effectively preventing depletion.

5.9 Future Work

5.9.1 Security

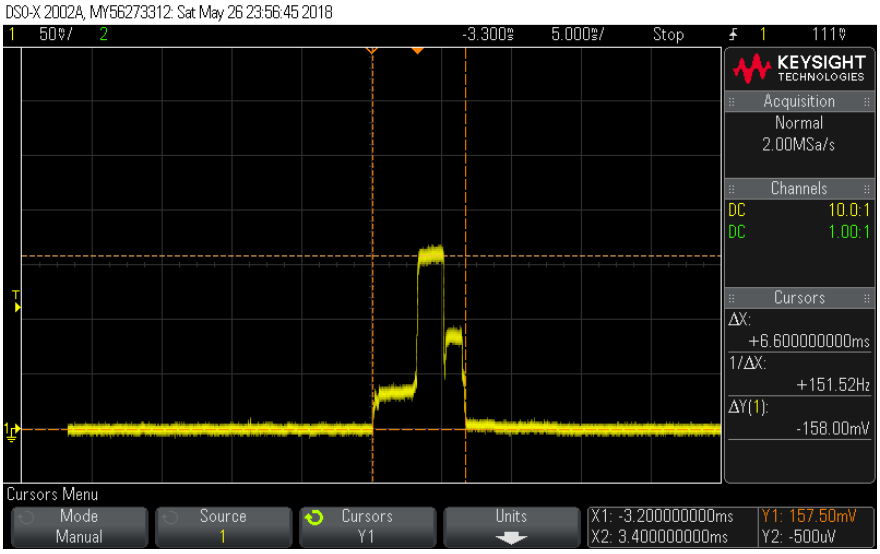
Very little security has been added in the WSNs developed throughout the project, which means that they may be easily hacked and the channels may be overheard. As the nodes do not have any actuators and simply read sensor values, this is not considered critical, however can be implemented as an improvement at a later point.

5.9.2 Optimize nRF52840 RAM Retention

Each $4KB$ RAM block in the nRF52840 consumes $30nA$ in sleep. There is a total of $256kB$ RAM, and therefore 64 blocks, which can be disabled if unused.

5.9.3 Alternative BME680 Operation

Figure 5.2 depicts measurements of the nRF52840 node with BME680 while not waiting for the sensor to measure and convert data. Instead, the MCU tells the BME680 to gather data while not collecting it until the next awake period. By doing this instead of the design presented and tested in the project, the awake period is significantly shortened, however the data measured by the BME680 is read by the MCU on the next transmission interval, effectively resulting in the environmental data having a delay of one transmission interval. This is a viable improvement where real-time measurements are of little importance.


DSO-X 2002A, MY56273312: Sat May 26 23:56:45 2018

1 50mV/ 2 -3.300ms 5.000ms/ Stop 1 111mV

Acquisition
Normal
2.00MSa/s

Channels
DC 10.0:1
DC 1.00:1

Cursors
ΔX: +6.600000000ms
1/ΔX: +151.52Hz
ΔY(1): -158.00mV

Cursors Menu
Mode Manual
Source 1
Cursors Y1
Units

X1: -3.200000000ms
X2: 3.400000000ms
Y1: 157.50mV
Y2: -500uV

Figure 5.2: Awake period when not waiting for BME680 data with nRF52840

5.9.4 OTA Firmware Update

Updating each node manually in a deployed sensor network that may consist of several hundreds of nodes would require an unreasonable amount of man-hours, and could prove to be impossible for some installations. The ability to perform OTA firmware updates would provide great advantages for a WSN.

5.9.5 Wake-up Radio

Multi-hop networks require several nodes to be on at the same time. This can be resolved with several methods, such as synchronous wake up based on a timer on each node, or with a Wake-up Radio. With Wake-up Radio, the nodes necessary to complete a transmission can be woken up only when there is something to send or forward, reducing power wasted. Wake-up radio may be useful in the WSNs for specific use-cases.

5.9.6 Research Alternative Energy Harvesting Sources

This far, EH with PV modules has proven to be sufficient in well lit office environments. If the nodes are to be placed in environments absent of light, other harvesting sources has to be further researched. Thermoelectric generators may provide power for in-wall installations, and development in triboelectric nanogenerators may improve harvestable energy available from vibrations.

Chapter 6

Conclusion

Two wireless sensor nodes based on different MCUs, namely the ATmega328p from Atmel and nRF52840 from Nordic Semiconductor, have been successfully developed and tested in both star and mesh topologies. The sensor nodes contain energy harvesting capabilities with the use of solar panels, allowing for continuous operation with no intervention for the entire lifetime of the components.

Two wireless network topologies are used, namely star and multi-hop topology, and custom protocols are developed for both. With star topology, both nodes consume less power than what is harvested even in a poor indoor light environment when using a $0.36W$ rated monocrystalline solar panel, and are able to operate for 7 to 12 months while using a rechargeable coin cell battery without any harvested energy. As for multi-hop topology, the results are mixed, as the ATmega328p consumes twice the amount of the energy harvested in a good light conditions in indoor environment. The nRF52840 in multi-hop topology, on the other hand, has proven functional for indoor use, assuming that it is not positioned in an environment with poor lighting. Both nodes needs to be tested thoroughly for outdoor environment.

The range of the ATmega328p node was measured to $228.5m$, while the nRF52840 obtained $1.8km$ while using coded transmissions. Considering this, as well as the lower power consumption of the nRF52840, it is concluded that the nRF52840 is superior to ATmega328p in all aspects with the exception of simplicity in firmware development. While the Arduino community provides libraries for a wide range of peripheral components, such as sensors and transceivers, the Nordic environment only provide simple examples on base designs, and no libraries for specific sensors are provided.

The nRF52840 node has been tested with both $1.9V$ and $3.3V$ supply. In star topology with the same peripheral components and firmware, the power consumption of the node with $1.9V$ was only 62% of the node with $3.3V$. Therefore, all nRF52840 based nodes were configured with 1.9V supply.

Testing proved that operation with supercapacitors instead of Li-Ion batteries is feasible to some extent. The supercapacitors have a higher self discharge rate and generally much lower capacities. The EH circuit is unable to charge the supercapacitors in good light conditions in indoor room when depleted, however after pre-charging the supercapacitor to 2V, the node worked well. Compared to the 7 to 12 months of operation with the battery and no energy harvesting, a fully charged 5F low leakage supercapacitor will last for 8 to 37 hours. Based on these findings, it is concluded that using a supercapacitor as energy storage is feasible for rooms equipped with good light while using the nRF52840 node. However, usage of supercapacitors are only recommended for outdoor use as the added benefit of sub-zero centigrade temperatures is not applicable for indoors.

In order to facilitate for IoT applications, gateways are developed for both WSNs. The gateways use ESP32 by Espressif, which has built-in Wi-Fi capabilities, in order to publish the sensor data via MQTT.

Bibliography

[1] “Normalized spectral response of a typical c-si solar cell,” last Accessed: 2018-06-03. [Online]. Available: https://www.researchgate.net/figure/232277326_fig1_Fig-3-Normalized-spectral-response-of-a-typical-c-Si-solar-cell-pyranometer-courtesy-of

[2] “Representative spectra of a white led, compact fluorescent vfl and incandescent bulbs,” last Accessed: 2018-06-03. [Online]. Available: https://www.researchgate.net/figure/263530987_fig1_Fig-1-Representative-spectra-of-a-White-LED-Compact-Fluorescent-CFL-and-Incandescent

[3] Atmel, “ATmega328p Datasheet,” last Accessed: 2018-06-03. [Online]. Available: http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

[4] “Arduino’s atmega328 power consumption,” last Accessed: 2018-06-03. [Online]. Available: https://www.gadgetmakersblog.com/power-consumption-arduinos-atmega328-microcontroller/

[5] T. Instruments, “Ultra low leakage switch,” last Accessed: 2018-06-03. [Online]. Available: http://www.ti.com/lit/ds/symlink/tps22860.pdf

[6] K. R. M. Kasemann, L. M. Reindl, “Photovoltaic energy harvesting under low lighting conditions,” last Accessed: 2018-06-03. [Online]. Available: https://www.researchgate.net/publication/257757787_Photovoltaic_Energy_Harvesting_under_Low_Lighting_Conditions

[7] “Bluetooth 5 core specifications,” last Accessed: 2018-06-03. [Online]. Available: https://www.bluetooth.com/specifications

[8] Z. Yu and K. Ogboenyira, “Renewable energy through micro-inverters,” last Accessed: 2018-06-03. [Online]. Available: http://www.powerelectronics.com/discrete-power-semis/renewable-energy-through-micro-inverters

[9] J. Jessen, M. Venzke, and V. Turau, "Design considerations for a universal smart energy module for energy harvesting in wireless sensor networks," last Accessed: 2018-06-03. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6086015

[10] "Avx scm series supercapacitors," last Accessed: 2018-06-03. [Online]. Available: http://www.avx.com/products/supercapacitors/scm-series/

[11] "Lir2450 datasheet," last Accessed: 2018-06-03. [Online]. Available: http://www.farnell.com/datasheets/1475807.pdf?_ga=2.175044267.845474401.1493834626-1387139467.1487680439

[12] A. Shrestha and L. Xing, "A performance comparison of different topologies for wireless sensor networks," last Accessed: 2018-06-03. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4227822

[13] L. Gu and J. A. Stankovic, "Radio-triggered wake-up for wireless sensor networks," Real-Time Systems, vol. 29, no. 2, pp. 157–182, Mar 2005, last Accessed: 2018-06-03. [Online]. Available: https://doi.org/10.1007/s11241-005-6883-z

[14] "Indirect health effects of relative humidity in indoor environments," last Accessed: 2018-06-03. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1474709/

[15] "Offices: Temperature and humidity - what are the 'rules'?" last Accessed: 2018-06-03. [Online]. Available: http://www.ohsrep.org.au/hazards/call-centres/offices-temperature-and-humidity-what-are-the-rules

[16] T. Vehviläinen et al., "High indoor CO2 concentrations in an office environment increases the transcutaneous CO2 level and sleepiness during cognitive work," last Accessed: 2018-06-03. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/15459624.2015.1076160?src=recsys&

[17] "Volatile Organic Compounds' Impact on Indoor Air Quality," last Accessed: 2018-06-03. [Online]. Available: https://www.epa.gov/indoor-air-quality-iaq/volatile-organic-compounds-impact-indoor-air-quality#Health_Effects

[18] "Human factors: Lighting, thermal comfort, working space, noise and vibration," last Accessed: 2018-06-03. [Online]. Available: http://www.hse.gov.uk/humanfactors/topics/lighting.htm#lighting

[19] "Bluetooth 5 range test with ti cc2640r2f," last Accessed: 2018-06-03. [Online]. Available: https://e2e.ti.com/blogs_/b/connecting_wirelessly/archive/2017/01/30/how-does-bluetooth-5-increase-the-achievable-range-of-a-bluetooth-low-energy-connection#

[20] “Bluetooth 5 mesh specification,” last Accessed: 2018-06-03. [Online]. Available: https://www.bluetooth.com/specifications/mesh-specifications

[21] “Basics of uart communication,” last Accessed: 2018-06-03. [Online]. Available: http://www.circuitbasics.com/basics-uart-communication/

[22] “Using the i2c bus,” last Accessed: 2018-06-03. [Online]. Available: http://www.robot-electronics.co.uk/i2c-tutorial

[23] “Serial peripheral interface (spi),” last Accessed: 2018-06-03. [Online]. Available: https://embeddedmicro.com/blogs/tutorials/serial-peripheral-interface-spi

[24] “Mqtt documentation,” last Accessed: 2018-06-03. [Online]. Available: http://mqtt.org/documentation

[25] “Hivemq - mqtt essentials part 6: Quality of service 0, 1 and 2,” last Accessed: 2018-06-03. [Online]. Available: http://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels

[26] “Mosquitto documentation,” last Accessed: 2018-06-03. [Online]. Available: https://mosquitto.org/documentation/

[27] “Arduino.” [Online]. Available: https://www.arduino.cc/

[28] “Altium Designer,” last Accessed: 2018-06-03. [Online]. Available: http://www.altium.com/altium-designer/

[29] “Segger Embedded Studio,” last Accessed: 2018-06-03. [Online]. Available: https://www.segger.com/products/development-tools/embedded-studio/

[30] “Autodesk Inventor,” last Accessed: 2018-06-03. [Online]. Available: https://www.autodesk.com/products/inventor/overview

[31] “Autodesk Fusion 360,” last Accessed: 2018-06-03. [Online]. Available: https://www.autodesk.com/products/fusion-360/overview

[32] “Tera Term,” last Accessed: 2018-06-03. [Online]. Available: https://ttssh2.osdn.jp/manual/en/

[33] “Hass.io,” last Accessed: 2018-06-03. [Online]. Available: https://www.home-assistant.io/hassio/

[34] “Fluke 115 multimeter datasheet,” last Accessed: 2018-06-03. [Online]. Available: http://en-us.fluke.com/products/digital-multimeters/fluke-115-digital-multimeter.html#techspecs

[35] "Keithley digital multimeter," last Accessed: 2018-06-03. [Online]. Available: http://www.tek.com/sites/tek.com/files/media/media/resources/2110%20DMM%20DataSht_HR.pdf

[36] "Keysight infiniivision dsox2002a oscilloscope specifications," last Accessed: 2018-06-03. [Online]. Available: https://www.keysight.com/en/pdx-x201827-pn-DSOX2002A/oscilloscope-70-mhz-2-analog-channels?pm=spc&nid=-32542.1150180&cc=NO&lc=eng

[37] "Keysight b2950a," last Accessed: 2018-06-03. [Online]. Available: https://www.keysight.com/en/pd-2441132-pn-B2985A/electrometer-high-resistance-meter-001fa?cc=CZ&lc=eng

[38] "CEL Robox Dual," last Accessed: 2018-06-03. [Online]. Available: http://cel-uk.com/3d-printer/rbx02.html

[39] "Snapmaker," last Accessed: 2018-06-03. [Online]. Available: https://snapmaker.com/product

[40] "Mooshimeter," last Accessed: 2018-06-03. [Online]. Available: https://moosh.im/mooshimeter/

[41] "Neonsee AAA Sun Simulator," last Accessed: 2018-06-03. [Online]. Available: http://www.neonsee.com/en/iv-measurement/

[42] T. Instruments, "Bq25570 nano power boost charger and buck converter for energy harvester powered applications," last Accessed: 2018-06-03. [Online]. Available: http://www.ti.com/lit/ds/symlink/bq25570.pdf

[43] "Bme680 datasheet," last Accessed: 2018-06-03. [Online]. Available: https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BME680-DS001-00.pdf

[44] "Hdc2010 datasheet," last Accessed: 2018-06-03. [Online]. Available: http://www.ti.com/lit/ds/symlink/hdc2010.pdf

[45] Arduino, "Arduino Nano User Manual," last Accessed: 2018-06-03. [Online]. Available: https://www.arduino.cc/en/uploads/Main/ArduinoNanoManual23.pdf

[46] ——, "Arduino Pro Mini User Manual," last Accessed: 2018-06-03. [Online]. Available: https://learn.sparkfun.com/tutorials/using-the-arduino-pro-mini-33v/all.pdf

[47] N. S. ASA, "Single chip 2.4 ghz transceiver," last Accessed: 2018-06-03. [Online]. Available: https://www.sparkfun.com/datasheets/Components/nRF24L01_prelim_prod_spec_1_2.pdf

[48] “Arduino low power - how to run atmega328p for a year on coin cell battery,” last Accessed: 2018-06-03. [Online]. Available: http://www.home-automation-community.com/arduino-low-power-how-to-run-atmega328p-for-a-year-on-coin-cell-battery/

[49] “Closedcube_bme680.h,” last Accessed: 2018-06-03. [Online]. Available: https://github.com/closedcube/ClosedCube_BME680_Arduino/blob/master/src/ClosedCube_BME680.h

[50] “Lowpower.h,” last Accessed: 2018-06-03. [Online]. Available: https://github.com/rocketscream/Low-Power

[51] “Brown-out detection,” last Accessed: 2018-06-03. [Online]. Available: https://scienceprog.com/microcontroller-brown-out-detection/

[52] “Rf24.h,” last Accessed: 2018-06-03. [Online]. Available: https://github.com/maniacbug/RF24

[53] “Esp32 datasheet,” last Accessed: 2018-06-03. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[54] “Mqtt pubsubclient by knolleary,” last Accessed: 2018-06-03. [Online]. Available: https://github.com/knolleary/pubsubclient

[55] “nrf52840 preview development kit.” [Online]. Available: http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52%2Fdita%2Fnrf52%2Fdevelopment%2Fnrf52840_pdk%2Fintro.html

[56] “Bmd-340 datasheet,” last Last Accessed: 2018-06-03. [Online]. Available: https://no.mouser.com/datasheet/2/883/BMD-340-DS_v0.8-1223159.pdf

[57] “Mcp73831t datasheet,” last Accessed: 2018-06-03. [Online]. Available: https://no.mouser.com/datasheet/2/268/20001984g-846362.pdf

Appendix A

Arduino Code

A.1 Star Code

Code A.1: Star Code

```
/*#########################################
  ######### Star Node #######################
  #########################################*/
#define MY_ID 17
#define DEBUG

#define GAS_HEAT_DURATION 200
#define GAS_HEAT_TEMP 300

#define ADC_SAMPLES 10
#define MAX_I2C_DEVICES 5

#define BME680_ADDR 0x77
#define HDC2010_ADDR 0x41
#define MAX_ADDR 0x4B

#ifdef DEBUG
#define SLEEP_TIME 2
#define GAS_READ_RATIO 1
#define DEBUG_SERIAL_BEGIN(x)  Serial.begin (x)
#define DEBUG_PRINT(x)  Serial.print (x)
#define DEBUG_PRINThex(x)  Serial.print (x,HEX)
#define DEBUG_PRINTln(x)  Serial.println (x)
#define DEBUG_FLUSH(x)  Serial.flush (x)
#else
#define SLEEP_TIME 10
#define GAS_READ_RATIO 10
#define DEBUG_SERIAL_BEGIN(x)
#define DEBUG_PRINT(x)
#define DEBUG_PRINThex(x)
#define DEBUG_PRINTln(x)
#define DEBUG_FLUSH(x)
#endif

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

```
#include "nRF24L01.h"
#include "RF24.h"
RF24 radio(9, 10);
#include "printf.h"
#include "LowPower.h"
#include "ClosedCube_BME680.h"
ClosedCube_BME680 bme680;

//nRF24 Variables
byte pipeNo;
uint64_t pipe =  0xf0f0f0f0e1;
bool ack = false;
uint64_t ackID32;
uint8_t radio_status, ackID = 0, ack_num;
bool channel_idle = false;

//I2C scan Variables
enum  {
  HDC2010_i2c,
  MAX44009_i2c,
  BME680_i2c
};
uint8_t i2c_device[MAX_I2C_DEVICES][2] = {
  {HDC2010_ADDR, false},
  {MAX_ADDR, false},
  {BME680_ADDR, false}
};
uint8_t devices[10];
uint8_t error;

//BME680 Variables
#ifdef BME680_ADDR
uint16_t bmedelay;
unsigned long bmeTime;
static uint8_t pres_msb, pres_lsb, pres_xlsb, temp_msb,
       temp_lsb, temp_xlsb, hum_msb, hum_lsb;
uint16_t bmePres;
float bmeTemp, bmeHum;
uint32_t bmeGas;
uint8_t bmeGasLvl;
bool GAS_ON = false;
uint8_t gas_counter = 10;
#endif

//HDC2010 Variables
unsigned long hdcTime;
#ifdef HDC2010_ADDR
float hdc2010_tempF, hdc2010_humF;
uint8_t hdc2010_data[4];
const double pow16 = pow(2, 16);
#endif

// MAX44009 Variables
uint8_t lux[2], lux_exponent;
uint32_t  maxLux;

//Other Variables
```

```

uint32_t wake_time, on_time, ack_time;
float bat;
uint16_t temp16[2], hum16[2];
boolean negative;
uint8_t failedTrans = 0;
uint8_t paAdjust = 0;
uint16_t pingCounter = 1;

struct mydata {
  uint8_t iddata;
  float tempdata;
  float humdata;
  uint32_t lightdata;
}; mydata data;
struct compData {
  uint16_t idLight;
  uint32_t humTemp;
  uint16_t bat;
  uint16_t pres;
  uint16_t gas;
}; compData compData;
struct sendData {
  uint8_t  id;
  uint8_t power_level;
  uint16_t light;
  uint16_t hum;
  int16_t temp;
  uint16_t bat;
  uint16_t pres;
  uint16_t gas;
  uint16_t packetNR;
}; sendData sendData;
void setup() {
  sendData.id = MY_ID;
  sendData.packetNR = 0;
  DEBUG_SERIAL_BEGIN(115200);
  pinMode(8, OUTPUT);
  pinMode(A0, INPUT);
  DEBUG_PRINT("Startup Node: ");
  DEBUG_PRINTln(MY_ID);
#ifdef DEBUG
  printf_begin();
#endif
  radio_init();
  scan_i2c_devices();
  if (i2c_device[BME680_i2c][1]) {
    bme680.init(BME680_ADDR); // I2C address: 0x76 or 0x77
    bme680.reset();
    bme680.setOversampling(BME680_OVERSAMPLING_X1,
                           BME680_OVERSAMPLING_X2, BME680_OVERSAMPLING_X16);
    bme680.setIIRFilter(BME680_FILTER_1); // 3
    delay(50);
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_init();
  }
  if (i2c_device[MAX44009_i2c][1]) {
    max_read();
  }
}
void loop() {
  on_time = millis();
  get_sensor_data();
  print_all();
  send_data();
  DEBUG_PRINT("On time = "); DEBUG_PRINTln((millis() - on_time));
  rtc_sleep(SLEEP_TIME);
}
...
```

A.1.1 BME680

Code A.2: BME680

```
...
void bme680_read() {
  bmeTemp = bme680.readTemperature();
  bmeHum = bme680.readHumidity();
  bmePres = bme680.readPressure();
  if (GAS_ON) {
    // wait for measurment bit
    delay(GAS_HEAT_DURATION + 5);
    bmeGas = bme680.readGasResistance();
    bme680.setGasOff();

    //bmeGasLvl = bme680_calc_iaq();
    gas_counter = 0;
    GAS_ON = false;
  }
}
...
```

A.1.2 HDC2010

Code A.3: HDC2010

```
...
void hdc2010_init() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x0E);
  Wire.write(0x00);
  Wire.endTransmission();
}
void hdc2010_ask_data() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x0F);
  Wire.write(0xA1);
  Wire.endTransmission();
}
void hdc2010_read_hum() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x02);
  Wire.endTransmission();
  Wire.requestFrom(HDC2010_ADDR, (uint8_t)2);
  hdc2010_data[2] = Wire.read();
  hdc2010_data[3] = Wire.read();
  Wire.endTransmission();
  hdc2010_humF = hdc2010_data[3] << 8 | hdc2010_data[2];
  hdc2010_humF = (hdc2010_humF / pow16) * 100;
}
void hdc2010_heater(uint8_t duration) {
  for (uint8_t i = 0; i < 3; i++) {
    Wire.beginTransmission(HDC2010_ADDR);
    Wire.write(0x0E);
    Wire.write(0x08);
    Wire.endTransmission();
    delay(duration);
    Wire.beginTransmission(HDC2010_ADDR);
    Wire.write(0x0E);
    Wire.write(0x00);
    Wire.endTransmission();
  }
}
void hdc2010_read() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x00);
  Wire.endTransmission();
  Wire.requestFrom(HDC2010_ADDR, (uint8_t)4);
  hdc2010_data[0] = Wire.read();
  hdc2010_data[1] = Wire.read();
  hdc2010_data[2] = Wire.read();
  hdc2010_data[3] = Wire.read();
  Wire.endTransmission();
  hdc2010_tempF = hdc2010_data[1] << 8 | hdc2010_data[0];
  hdc2010_tempF = ((hdc2010_tempF / pow16) * 165  - 40);

  hdc2010_humF = hdc2010_data[3] << 8 | hdc2010_data[2];
  hdc2010_humF = (hdc2010_humF / pow16) * 100;
}
...
```

A.1.3 MAX44009

Code A.4: MAX44009

```
...
void max_read() {
  Wire.beginTransmission(MAX_ADDR);
  Wire.write(0x03);
  Wire.requestFrom(MAX_ADDR, (uint8_t)2);
  lux[0] = Wire.read();
  lux[1] = Wire.read();
  Wire.endTransmission();

  lux_exponent    = ((lux[0] >> 4) & 0x0F);
  lux[0]          = ((lux[0] << 4) & 0xF0);
  lux[1]         &= 0x0F;

  //lux_value    = 0.045 * ( lux_high + lux_low ) * (1<< lux_exponent);
  maxLux       = 45L * ( lux[0] | lux[1] ) * (1 << lux_exponent);
  maxLux = maxLux / 1000;
}
...
```

```
void rtc_sleep(uint16_t sleeptime) {
  DEBUG_PRINT("Going to sleep for: ");
  DEBUG_PRINT(sleeptime);
  DEBUG_PRINTln(" seconds\n");
  //delay(3);
  DEBUG_FLUSH();
  uint16_t remainder = sleeptime;
  uint8_t cycles;
  while (remainder != 0) {
    if (remainder >= 8) {
      cycles = remainder / 8;
      remainder = remainder % 8;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF);
      }
    }
    else if (remainder >= 4) {
      cycles = remainder / 4;
      remainder = remainder % 4;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF);
      }
    }
    else if (remainder >= 2) {
      cycles = remainder / 2;
      remainder = remainder % 2;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
      }
    }
    else if (remainder >= 1) {
      cycles = remainder / 1;
      remainder = remainder % 1;
      for (uint8_t i = 0; i < cycles; i++) {
        LowPower.powerDown(SLEEP_1S, ADC_OFF, BOD_OFF);
      }
    }
    else {
      remainder = 0;
    }
  }
}
void get_sensor_data() {
  digitalWrite(8, HIGH);
  sendData.packetNR++;
  if (i2c_device[BME680_i2c][1]) {
    bmedelay = 14;
    gas_counter++;
    if (gas_counter >= GAS_READ_RATIO) {
      GAS_ON = true;
      bmedelay = bmedelay + GAS_HEAT_DURATION + 5;
    }
    if (GAS_ON) {
      bme680.setGasOn(GAS_HEAT_TEMP, GAS_HEAT_DURATION);
    }
    bme680.setForcedMode(); bmeTime = millis();
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_ask_data(); hdcTime = micros();
  }
  if (i2c_device[MAX44009_i2c][1]) {
    max_read();
    sendData.light = maxLux;
  }
#ifdef ADC_SAMPLES
  for (int i = 0; i < ADC_SAMPLES; i++) {
    bat = bat + analogRead(A0);
  }
  bat = bat / ADC_SAMPLES;
  bat = ((bat / 1024) * 3.3);
  bat = bat * 2;
  sendData.bat = bat * 100;
  digitalWrite(8, LOW);
#endif
  if (i2c_device[HDC2010_i2c][1]) {
    while ((hdcTime + 600) > (micros())) {} //wait for hdc data
    hdc2010_read();
    hdc2010_tempF = hdc2010_tempF * 100;
    sendData.temp = hdc2010_tempF;
    hdc2010_humF = hdc2010_humF * 100;
    sendData.hum = hdc2010_humF;
  }
  if (i2c_device[BME680_i2c][1]) {
    while ((bmeTime + bmedelay) > (millis())) {} //wait for bme data
    bme680_read();
    bmeTemp = bmeTemp * 100;
    sendData.temp = bmeTemp;
    bmeHum = bmeHum * 100;
    sendData.hum = bmeHum;
    sendData.gas = bmeGas / 10000; //bmeGas;// on_time;
    sendData.pres = bmePres;
  }
  print_all();
}
void scan_i2c_devices() {
  DEBUG_PRINTln("I2C scan:");
  for (uint8_t address = 0; address < 125; address++) {
    Wire.beginTransmission(address);
    error = Wire.endTransmission();
    if (error == 0) {
      for (uint8_t i = 0; i < MAX_I2C_DEVICES; i++) {
        if (address == i2c_device[i][0]) {
          i2c_device[i][1] = true;
        }
      }
      DEBUG_PRINT(F("Found device: 0x"));
      DEBUG_PRINThex(address);
      DEBUG_PRINTln();
    }
  }
  DEBUG_PRINTln();
}
```

A.2 Multi-hop Code

Code A.5: Multi-hop Code

```
/*############################################
  ####### Multi-hop Node ####################
  ##########################################*/
#define NODE 50
#define DEBUG
#define SEND_INTERVAL 3 // Multiples of 2s
#define GAS_HEAT_DURATION 200 // 200 //  in millisec. from 63-4032
#define GAS_HEAT_TEMP 300 //200 // from 200-400c
#define GAS_READ_RATIO 10
#define ADC_SAMPLES 10
#define MAX_I2C_DEVICES 5
#define BME680_ADDR 0x77
#define HDC2010_ADDR 0x41
#define MAX_ADDR 0x4B

#ifdef DEBUG
#define DEBUG_SERIAL_BEGIN(x)  Serial.begin (x)
#define DEBUG_PRINT(x)  Serial.print (x)
#define DEBUG_PRINThex(x)  Serial.print (x,HEX)
#define DEBUG_PRINTln(x)  Serial.println (x)
#define DEBUG_FLUSH(x)  Serial.flush (x)
#else
#define DEBUG_SERIAL_BEGIN(x)
#define DEBUG_PRINT(x)
#define DEBUG_PRINThex(x)
#define DEBUG_PRINTln(x)
#define DEBUG_FLUSH(x)
#endif

#include <Wire.h>
#include <SPI.h>
#include "nRF24L01.h"
#include "RF24.h"
RF24 radio(9, 10);
#include "printf.h"
#include "LowPower.h"
#include <Adafruit_Sensor.h>
#include "ClosedCube_BME680.h"
ClosedCube_BME680 bme680;

//following is used as ID's and "pipes"
#define master 0x00
#define br_addr 0xff //Broadcast address
#define max_neighbors 10

//I2C scan Variables
enum {
  HDC2010_i2c,
  MAX44009_i2c,
  BME680_i2c
};
uint8_t i2c_device[MAX_I2C_DEVICES][2] = {
  {HDC2010_ADDR, false},
  {MAX_ADDR, false},
  {BME680_ADDR, false}
};
uint8_t devices[10];
uint8_t error;

//BME680 Variables
#ifdef BME680_ADDR
uint16_t bmedelay;
unsigned long bmeTime;
uint16_t bmePres;
float bmeTemp, bmeHum;
uint32_t bmeGas;
uint8_t bmeGasLvl;
bool GAS_ON = false;
uint8_t gas_counter = 10;
#endif

//HDC2010 Variables
#ifdef HDC2010_ADDR
unsigned long hdcTime;
float hdc2010_tempF, hdc2010_humF;
uint8_t hdc2010_data[4];
const double pow16 = pow(2, 16);
#endif

// MAX44009 Variables
#ifdef MAX_ADDR
uint8_t lux[2], lux_exponent;
uint32_t  maxLux;
#endif
//Other Variables
float bat;
uint16_t temp16[2], hum16[2];
boolean negative;
uint8_t failedTrans = 0;
uint8_t paAdjust = 0;
uint16_t pingCounter = 1;

typedef struct data_msg {
  uint8_t  destID;
  uint8_t sourceID;
  uint8_t hops;
  uint16_t packetNR;
  uint16_t light;
  uint16_t hum;
  uint16_t temp;
  uint16_t bat;
  uint16_t pres;
  uint16_t gas;
}; data_msg msg;

```

```
typedef struct br_msg {
  uint8_t id;
  uint8_t hops;
}; br_msg wake_msg;

unsigned long current_millis, timeout_millis, relaytime_millis, sink_timer = 0;
unsigned long ack_micros;

// Radio variables
uint64_t pipe =  0xf0f0f0f0e1;
uint16_t packetNR = 0;
uint8_t my_hops_to_sink = 254;
uint8_t send_counter = 0;
bool channel_idle = false;
bool ack = false;

void setup() {
  DEBUG_SERIAL_BEGIN(57600);
  initRadio();
  pinMode(8, OUTPUT);
  pinMode(A0, INPUT);
  scan_i2c_devices();
  if (i2c_device[BME680_i2c][1]) {
    bme680.init(BME680_ADDR); // I2C address: 0x76 or 0x77
    bme680.reset();
    bme680.setOversampling(BME680_OVERSAMPLING_X1, BME680_OVERSAMPLING_X2, BME680_OVERSAMPLING_X16);// 1,2,16
    bme680.setIIRFilter(BME680_FILTER_1); // 3
    delay(50);
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_init();
    //hdc2010_heater();
  }
  if (i2c_device[MAX44009_i2c][1]) {
    max_read();
  }
  DEBUG_FLUSH();
}
void loop() {
  if (send_counter < SEND_INTERVAL) {
    wake_ping();
  }
  else {
    get_sensor_data();
    wait_to_send();
    send_counter = 0;
  }
  radio.powerDown();
  LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
  radio.powerUp();
  send_counter++;
}
void wake_ping() {
  DEBUG_FLUSH();
  wake_msg.id = NODE;
  wake_msg.hops = my_hops_to_sink;
  radio.stopListening();
  carrier_sense();
  radio.write(&wake_msg, sizeof(wake_msg));
  radio.startListening();
  wait_for_response();
}
void send_ack() {
  radio.stopListening();
  delay(1);
  radio.write(&msg.sourceID, sizeof(msg.sourceID));
  radio.startListening();
}
void wait_for_ack(uint8_t sourceID) {
  ack = false;
  uint8_t recByte;
  ack_micros = micros();
  while ( micros() - ack_micros < 3000) {
    if (radio.available()) {
      if (radio.getDynamicPayloadSize() == sizeof(recByte)) {
        radio.read(&recByte, sizeof(recByte));
        if (recByte == sourceID) {
          ack = true;
        }
      }
    }
  }
}
void send_packet() {
  radio.stopListening();
  if (!radio.write(&msg, sizeof(msg))) {
    DEBUG_PRINTln("TX failed...");
    DEBUG_FLUSH();
  }
  radio.startListening();
}
void carrier_sense() {
  channel_idle = false;
  radio.startListening();
  delayMicroseconds(128);
  radio.stopListening();
  if (!radio.testCarrier() ) {
    DEBUG_PRINTln("Channel IDLE");
    channel_idle = true;
  }
  else {
    DEBUG_PRINTln("BUSY");
  }
}
void print_data() {
  DEBUG_PRINT("Message received with source: "); DEBUG_PRINTln(msg.sourceID);
  DEBUG_PRINT("Hops = "); DEBUG_PRINTln(msg.hops);
  DEBUG_PRINT("Temperature = "); DEBUG_PRINTln(msg.temp);
  DEBUG_PRINT("Humidity = "); DEBUG_PRINTln(msg.hum);
  DEBUG_PRINT("Battery = "); DEBUG_PRINTln(msg.bat);
  DEBUG_FLUSH();
}
void bme680_read() {
  bmeTemp = bme680.readTemperature();
  bmeHum = bme680.readHumidity();
  bmePres = bme680.readPressure();
  if (GAS_ON) {
    delay(GAS_HEAT_DURATION+5);
    bmeGas = bme680.readGasResistance();
    bme680.setGasOff();
    gas_counter = 0;
    GAS_ON = false;
  }
}
void hdc2010_init() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x0E);
  Wire.write(0x00);
  Wire.endTransmission();
}
void hdc2010_ask_data() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x0F);
  Wire.write(0xA1);
  Wire.endTransmission();
}
void hdc2010_read_hum() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x02);
  Wire.endTransmission();
  Wire.requestFrom(HDC2010_ADDR, (uint8_t)2);
  hdc2010_data[2] = Wire.read();
  hdc2010_data[3] = Wire.read();
  Wire.endTransmission();
  hdc2010_humF = hdc2010_data[3] << 8 | hdc2010_data[2];
  hdc2010_humF = (hdc2010_humF / pow16) * 100;
}
void hdc2010_heater() {
  for (uint8_t i = 0; i < 3; i++) {
    Wire.beginTransmission(HDC2010_ADDR);
    Wire.write(0x0E);
    Wire.write(0x08);
    Wire.endTransmission();
    delay(10);
    Wire.beginTransmission(HDC2010_ADDR);
    Wire.write(0x0E);
    Wire.write(0x00);
    Wire.endTransmission();
    hdc2010_ask_data();
    delay(30);
    hdc2010_read();
    //hdc2010_read_hum();
    DEBUG_PRINT(F("HDC2010Temp: "));
    DEBUG_PRINTln(msg.temp);
    DEBUG_PRINT(F("HDC2010Hum: "));
    DEBUG_PRINTln(msg.hum);
    delay(1000);
  }
}
void hdc2010_read() {
  Wire.beginTransmission(HDC2010_ADDR);
  Wire.write(0x00);
  Wire.endTransmission();
  Wire.requestFrom(HDC2010_ADDR, (uint8_t)4);
  hdc2010_data[0] = Wire.read();
  hdc2010_data[1] = Wire.read();
  hdc2010_data[2] = Wire.read();
  hdc2010_data[3] = Wire.read();
  Wire.endTransmission();
  hdc2010_tempF = hdc2010_data[1] << 8 | hdc2010_data[0];
  hdc2010_tempF = ((hdc2010_tempF / pow16) * 165  - 40);

  hdc2010_humF = hdc2010_data[3] << 8 | hdc2010_data[2];
  hdc2010_humF = (hdc2010_humF / pow16) * 100;
}
void scan_i2c_devices() {
  DEBUG_PRINTln("I2C scan:");
  for (uint8_t address = 0; address < 125; address++) {
    Wire.beginTransmission(address);
    error = Wire.endTransmission();
    if (error == 0) {
      for (uint8_t i = 0; i < MAX_I2C_DEVICES; i++) {
        if (address == i2c_device[i][0]) {
          i2c_device[i][1] = true;
        }
      }
      DEBUG_PRINT(F("Found device: 0x"));
      DEBUG_PRINThex(address);
      DEBUG_PRINTln();
    }
  }
  DEBUG_PRINTln();
}
void max_read() {
  Wire.beginTransmission(MAX_ADDR);
  Wire.write(0x03);
  Wire.requestFrom(MAX_ADDR, (uint8_t)2);
  lux[0] = Wire.read();
  lux[1] = Wire.read();
  Wire.endTransmission();

  lux_exponent    = ((lux[0] >> 4) & 0x0F);
  lux[0]          = ((lux[0] << 4) & 0xF0);
  lux[1]         &= 0x0F;

  //lux_value    = 0.045 * ( lux_high + lux_low ) * (1<< lux_exponent);
  maxLux       = 45L * ( lux[0] | lux[1] ) * (1 << lux_exponent);
  maxLux = maxLux / 1000;
}
void get_sensor_data() {
  packetNR++;
  digitalWrite(8, HIGH);
  msg.packetNR = packetNR;
  msg.sourceID = NODE;
  msg.hops = 0;
  if (i2c_device[BME680_i2c][1]) {
    bmedelay = 14;
    gas_counter++;
    if (gas_counter >= GAS_READ_RATIO) {
      GAS_ON = true;
      bmedelay = bmedelay + GAS_HEAT_DURATION + 5;
    }
    if (GAS_ON) {
      bme680.setGasOn(GAS_HEAT_TEMP, GAS_HEAT_DURATION);
    }
    bme680.setForcedMode(); bmeTime = millis();
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_ask_data(); hdcTime = micros();
  }
  if (i2c_device[MAX44009_i2c][1]) {
    max_read();
    msg.light = maxLux;
  }
#ifdef ADC_SAMPLES
  for (int i = 0; i < ADC_SAMPLES; i++) {
    bat = bat + analogRead(A0);
  }
  bat = ((bat / ADC_SAMPLES) / 1024) * 3.3 * 2;
  msg.bat = bat * 100;
  digitalWrite(8, LOW);
#endif
  if (i2c_device[HDC2010_i2c][1]) {
    while ((hdcTime + 300) > (micros())) {} //wait for hdc data
    hdc2010_read();
    hdc2010_tempF = hdc2010_tempF * 100;
    msg.temp = hdc2010_tempF;
    hdc2010_humF = hdc2010_humF * 100;
    msg.hum = hdc2010_humF;
  }
  if (i2c_device[BME680_i2c][1]) {
    while ((bmeTime + bmedelay) > (millis())) {} //wait for bme data
    bme680_read();
    msg.temp = bmeTemp * 100;
    msg.hum = bmeHum * 100;
    msg.gas = bmeGas / 10000; //bmeGas;// on_time;
    msg.pres = bmePres;
  }
}
void initRadio() {
  radio.begin();
  radio.setCRCLength(RF24_CRC_16);
  radio.setPALevel(RF24_PA_MIN);
  radio.setDataRate(RF24_1MBPS);
  radio.setChannel(124);
  radio.enableDynamicPayloads();
  radio.setRetries(0, 0);
  radio.setAutoAck(false);
  radio.openWritingPipe(pipe);
  radio.openReadingPipe(1, pipe);
  radio.powerDown();
}
void wait_for_response() {
  current_millis = millis();
  while ( millis() - current_millis < 10) { // 7 lowest working
```

```
    if (radio.available()) {
      if (radio.getDynamicPayloadSize() == sizeof(msg)) {
        radio.read(&msg, sizeof(msg));
        send_ack();
        DEBUG_PRINT("Message received with source: "); DEBUG_PRINTln(msg.sourceID);
        DEBUG_PRINT("Hops = "); DEBUG_PRINTln(msg.hops);
        DEBUG_PRINT("Temperature = "); DEBUG_PRINTln(msg.temp);
        DEBUG_PRINT("Humidity = "); DEBUG_PRINTln(msg.hum);
        DEBUG_PRINT("Battery = "); DEBUG_PRINTln(msg.bat);
        DEBUG_FLUSH();
        msg.hops++;
        wait_to_send();
        break;
      }
      if (!(radio.getDynamicPayloadSize() == sizeof(msg))) {
        radio.flush_rx();
      }
    }
  }
  radio.stopListening();
}
void wait_to_send() {
  timeout_millis =  millis();
  if (my_hops_to_sink == 1) { // Sink is my next hop, send now
    radio.stopListening();
    channel_idle = false;
    ack = false;
    uint8_t retry = 6;
    while (!ack) {
      while (!channel_idle) {
        carrier_sense();
      }
      send_packet();
      wait_for_ack(msg.sourceID);
      retry--;
      if (!retry) {
        my_hops_to_sink++;
        break;
      }
    }
  }
  while (my_hops_to_sink > 1) {
    if (radio.available()) {
      if (radio.getDynamicPayloadSize() == sizeof(wake_msg)) {
        radio.read(&wake_msg, sizeof(wake_msg));
        DEBUG_PRINT("ID = ");
        DEBUG_PRINTln(msg.sourceID);
        DEBUG_FLUSH();
        if (wake_msg.hops < my_hops_to_sink) {
          my_hops_to_sink = wake_msg.hops + 1;
          msg.destID = wake_msg.id;
          delayMicroseconds(100 * random(1, 4)); //random delay to avoid collisions
          DEBUG_PRINT("Waited: ");
          DEBUG_PRINTln(millis() - timeout_millis);
          DEBUG_FLUSH();
          DEBUG_PRINT("Sending data to ");
          DEBUG_PRINThex(msg.sourceID);
          DEBUG_PRINTln();
          DEBUG_FLUSH();
          radio.stopListening();
          carrier_sense();
          if (channel_idle) {
            send_packet();
            wait_for_ack(msg.sourceID);
            if (ack) {
              break;
            }
            else {
              timeout_millis = millis();
            }
          }
          else {
            timeout_millis = millis();
          }
        }
      }
      else if (radio.getDynamicPayloadSize() != sizeof(wake_msg)) {
        radio.flush_rx();
      }
    }
    if ((millis() - timeout_millis > 2200) && (my_hops_to_sink < 254)) {
      my_hops_to_sink++;
      send_counter++;
      timeout_millis = millis();
      DEBUG_PRINT("Timeout! Hops = ");
      DEBUG_PRINTln(my_hops_to_sink);
      DEBUG_FLUSH();
    }
  }
  wake_ping();
}
```

A.3 ESP32 with nRF24l01 Gateway

Code A.6: ESP32 with nRF24l01 Gateway

```
/*##########################################################
  ##### ESP32 W/nRF24l01+ Star/Multi-hop Gateway #########
  ##########################################################*/
#include <WiFi.h>
#include <PubSubClient.h>
#include "RF24.h"
RF24 radio(16, 17);
//#define MULTIHOP_GATEWAY
//#define DEBUG

#ifdef DEBUG
#define DEBUG_SERIAL_BEGIN(x)  Serial.begin (x)
#define DEBUG_PRINT(x)  Serial.print (x)
#define DEBUG_PRINTln(x)  Serial.println (x)
#else
#define DEBUG_SERIAL_BEGIN(x)
#define DEBUG_PRINT(x)
```

```
#define DEBUG_PRINTln(x)
#endif

#define mqtt_server "192.168.0.10"
#define mqtt_user "pi"
#define mqtt_password "raspberry"
uint16_t lastpacket[60];
uint8_t recArray[20];
struct recData {
  uint8_t  destID;
  uint8_t sourceID;
  uint8_t PA_OR_HOPS;
  uint16_t packetNR;
  uint16_t light;
  uint16_t hum;
  int16_t temp;
  uint16_t bat;
  uint16_t pres;
  uint16_t gas;
}; recData recData;
typedef struct br_msg {
  uint8_t id;
  uint8_t hops;
}; br_msg wake_msg;
//nRF24 Variables
byte pipeNo;
uint64_t pipe =  0xf0f0f0f0e1;
bool ack = false;
uint8_t radio_status, ackID = 0, ack_num;
char *topicBuf = (char*)malloc(10);
char *varBuf = (char*)malloc(35);
const char *topicConst;
const char *varConst;
const char* ssid     = "dlink-8410";
const char* password = "vkwix58592";

WiFiClient espClient;
PubSubClient client(espClient);

uint16_t temp16, hum16, lux16;
uint8_t data[20];
uint8_t byteNR;
long lastMsg = 0;
float temp = 5;
float hum = 5;
float bat = 5;
unsigned long wake_ping_timer;

void setup_wifi() {
  delay(10);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  DEBUG_PRINT("Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    DEBUG_PRINT(".");
  }
  DEBUG_PRINTln("Connected");
}
void setup()
{
  DEBUG_SERIAL_BEGIN(115200);
  DEBUG_PRINTln("start");
  delay(10);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  reconnect();
  radio_init();
  wake_msg.id = 0;
  wake_msg.hops = 0;
#ifdef MULTIHOP_GATEWAY
  wake_ping_timer = millis();
#endif
}
void loop()
{
  if (radio.available()) {
    if (!(radio.getDynamicPayloadSize() == sizeof(recData))) {//not correct packet
      radio.flush_rx();
    }
    else { // Received data packet
      radio.read(&recArray, sizeof(recData));
      if (lastpacket[recData.sourceID] == recData.packetNR) {
        //Dont publish duplicate
        send_ack();
      } else {
        send_ack();
        lastpacket[recData.sourceID] =  recData.packetNR;
        publish_mqtt();
      }
    }
  }
#ifdef MULTIHOP_GATEWAY
  if ((wake_ping_timer + 1500) < millis()) {
    radio.stopListening();
    radio.write(&wake_msg, sizeof(wake_msg));
    radio.startListening();
    wake_ping_timer = millis();
  }
#endif
}
void publish_mqtt() {
  DEBUG_PRINT("######Sensor: ");
  DEBUG_PRINT(String(recData.id).c_str());
  DEBUG_PRINTln("######");
  DEBUG_PRINT("Power level: ");
  DEBUG_PRINTln(recData.power_level);
  temp = recData.temp;
  temp = temp / 100;
  hum = recData.hum;
  hum = hum / 100;
  lux16 = recData.light;
  bat = recData.bat;
  bat = bat / 100;
  DEBUG_PRINT("New lux: ");
  DEBUG_PRINTln(String(recData.light).c_str());
  DEBUG_PRINT("New temperature: ");
  DEBUG_PRINTln(String(temp).c_str());
  DEBUG_PRINT("New humidity: ");
  DEBUG_PRINTln(hum);
  DEBUG_PRINT("New bat: ");
  DEBUG_PRINTln(recData.bat);
  DEBUG_PRINT("Packet NR: ");
  DEBUG_PRINTln(recData.packetNR);
  if (WiFi.status() != WL_CONNECTED) {
    setup_wifi();
  }
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
  delay(10);

  sprintf(topicBuf, "node%i/", recData.sourceID);
  sprintf(varBuf, "%i,%.2f,%.2f,%i,%.2f,%i,%i,%i,%i",
          recData.sourceID, temp, hum, recData.light, bat, recData.pres,
          recData.gas, recData.packetNR, recData.PA_OR_HOPS);
  topicConst = topicBuf;
  varConst = varBuf;
  client.publish(topicConst, varConst, true);
}
void reconnect() {
  // Loop until we're reconnected
  DEBUG_PRINT("MQTT");
  while (!client.connected()) {
    //  DEBUG_PRINT("Attempting MQTT connection...");
    if (client.connect("ESP8266Client", mqtt_user, mqtt_password)) {
      DEBUG_PRINTln("connected");
    } else {
      DEBUG_PRINT(client.state());
    }
  }
}
void radio_init() {
  radio.begin();
  radio.setCRCLength(RF24_CRC_16);
  radio.setPALevel(RF24_PA_MAX);
  radio.setDataRate(RF24_1MBPS);
  radio.setChannel(0x76);
  radio.enableDynamicPayloads();
  radio.setRetries(0, 0);
  radio.setAutoAck(false);
  radio.openWritingPipe(pipe);
  radio.openReadingPipe(1, pipe);
  radio.startListening();
}
void send_ack() {
#ifndef MULTIHOP_GATEWAY
  if ((recData.PA_OR_HOPS >= 0) && (recData.PA_OR_HOPS <= 3)) {
    radio.setPALevel(recData.PA_OR_HOPS);
  }

  DEBUG_PRINT("ack PA level: ");
  DEBUG_PRINTln(recData.PA_OR_HOPS);
#endif
  delayMicroseconds(2500);
  radio.stopListening();
  radio.write(&recData.sourceID, sizeof(recData.sourceID));
  radio.startListening();

}
```

A.4 ESP32 with nRF52840 gateway

Code A.7: ESP32 with nRF52840 gateway

```
/*###########################################
  #######ESP32 /w nRF52840 Gateway #########
  ###########################################*/
#include <WiFi.h>
#include <PubSubClient.h>
#include "HardwareSerial.h"
HardwareSerial Serial2(2);
#define DEBUG

#ifdef DEBUG
#define DEBUG_SERIAL_BEGIN(x)  Serial.begin (x)
#define DEBUG_PRINT(x)  Serial.print (x)
#define DEBUG_PRINTln(x)  Serial.println (x)
#else
#define DEBUG_SERIAL_BEGIN(x)
#define DEBUG_PRINT(x)
#define DEBUG_PRINTln(x)
#endif

#define mqtt_server "m23.cloudmqtt.com"
#define mqtt_user "qqzxyeqx"
#define mqtt_password "9P_dFROEu3JP"

char *topicBuf = (char*)malloc(20);
char *varBuf = (char*)malloc(40);
const char *topicConst;
const char *varConst;

const char* ssid     = "dlink-8410";
const char* password = "vkwix58592";

WiFiClient espClient;
PubSubClient client(espClient);
unsigned long client_timer;
uint16_t light16, hum16, bat16, pres16, gas16, packetNR;
int16_t temp16;
float tempF, humF, batF;
uint8_t bat;
uint8_t discard;;
uint8_t data[20];
uint8_t byteNR;
```

```
long lastMsg = 0;
float temp = 5;
float hum = 5;
int id = 1;
int lux;
byte firstByte;

void setup_wifi() {
  delay(10);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  DEBUG_PRINT("Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    DEBUG_PRINT(".");
  }
  DEBUG_PRINTln("Connected");
  //Serial2.flush();
}
void setup()
{
  DEBUG_SERIAL_BEGIN(115200);
  Serial2.begin(115200, SERIAL_8N1, 16, 17);
  DEBUG_PRINTln("start");
  delay(10);
  setup_wifi();
  client.setServer(mqtt_server, 12039); // 1883
  reconnect();
}
void loop()
{
  if ((Serial2.available() > 7)) { // wait for 6 bytes or timeout
    if (Serial2.read() == 155) {
      data[0] = Serial2.read();
      for (int i = 1; i < data[0] + 2; i++) {
        data[i] = Serial2.read();
      }
      while (Serial2.available()) {
        Serial2.read();
      }
      publish_mqtt();
      for (int i = 0; i < 20; i++) {
        data[i] = 0;
      }
    }
  }
}
void publish_mqtt() {
  DEBUG_PRINTln("Got data!");

  data[3] = data[3]; //ID
  data[4] = data[4]; // PLVL for star / hops for multihop
  packetNR = data[5] << 8 | data[6];
  bat16 = data[7] << 8 | data[8];
  batF = bat16;
  batF = batF / 1000;
  light16 = data[9] << 8 | data[10];
  temp16 = data[11] << 8 | data[12];
  tempF = temp16;
  tempF = tempF / 100;
  hum16 = data[13] << 8 | data[14];
  humF = hum16;
  humF = humF / 100;
  pres16 = data[15] << 8 | data[16];
  gas16 = data[17] << 8 | data[18];
  // Send sensor data to Raspberry Pi server //

  DEBUG_PRINT("####Sensor: ");
  DEBUG_PRINT(data[3]);
  DEBUG_PRINTln("####");
  DEBUG_PRINT("Power Level: ");
  DEBUG_PRINTln(data[4]);
  DEBUG_PRINT("PacketNR: ");
  DEBUG_PRINTln(packetNR);
  DEBUG_PRINT("Battery: ");
  DEBUG_PRINTln(bat16);
  DEBUG_PRINT("Light: ");
  DEBUG_PRINTln(light16);
  DEBUG_PRINT("Temperature: ");
  DEBUG_PRINTln(temp16 / 100);
  DEBUG_PRINT("Humidity: ");
  DEBUG_PRINTln(hum16 / 100);
  DEBUG_PRINT("Pressure: ");
  DEBUG_PRINTln(pres16);
  DEBUG_PRINT("Gas: ");
  DEBUG_PRINTln(gas16);

  if (WiFi.status() != WL_CONNECTED) {
    setup_wifi();
  }
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
  delay(10);
  if ((pres16 != 0) && (gas16 != 0)) {
    sprintf(topicBuf, "node%i/", data[3]);
    sprintf(varBuf, "%i,%.2f,%.2f,%i,%.2f,%i,%i,%i,%i", data[3],
            tempF, humF, light16, batF, pres16, gas16, packetNR, data[4]);
    topicConst = topicBuf;
    varConst = varBuf;
    client.publish(topicConst, varConst, true);
  } else {
    sprintf(topicBuf, "node%i/", data[3]);
    sprintf(varBuf, "%i,%.2f,%.2f,%i,%.2f,%s,%s,%i,%i", data[3],
            tempF, humF, light16, batF, "N/A", "N/A", packetNR, data[4]);
    topicConst = topicBuf;
    varConst = varBuf;
    client.publish(topicConst, varConst, true);
  }
}
void reconnect() {
  // Loop until we're reconnected
  DEBUG_PRINT("MQTT");
  while (!client.connected()) {
    //    DEBUG_PRINT("Attempting MQTT connection...");
    if (client.connect("ESP8266Client", mqtt_user, mqtt_password)) {
      DEBUG_PRINTln("connected");
    } else {
    }
  }
}
```

Appendix B

nRF52840 Code

B.1 Star Node

Code B.1: Star Code

```
/*##########################################
  ##### Star Node #########################
  ##########################################*/
#define NRF_LOG_ENABLED 1 // enable/disable prints
#define NODE 112
#define START_PA 10 // Initial power level
#define ENABLE_PIN NRF_GPIO_PIN_MAP(1, 8) // Battery sensing enable pin
#define SEND_INTERVAL APP_TIMER_TICKS(60 * 1000)
#define GAS_READ_RATIO 10
#define HDC2010_ADDR (0x41)
#define MAX_ADDR (0x4b)
#define RSSI_LOWER_LIMIT 85
#define RSSI_UPPER_LIMIT 75
#define SAMPLES_IN_BUFFER 1
#define ADC_FACTOR 5390.625 // Precalculated ADC factor

#include "app_error.h"
#include "app_timer.h"
#include "boards.h"
#include "bsp.h"
#include "nordic_common.h"
#include "nrf_bme680.h"
#include "nrf_delay.h"
#include "nrf_drv_clock.h"
#include "nrf_drv_ppi.h"
#include "nrf_drv_saadc.h"
#include "nrf_drv_twi.h"
#include "nrf_error.h"
#include "nrf_gpio.h"
#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
```

```
#include <string.h>

#ifndef TWI_INSTANCE_ID
#define TWI_INSTANCE_ID 0
static const nrf_drv_twi_t m_twi = NRF_DRV_TWI_INSTANCE(TWI_INSTANCE_ID);
#endif
//Battery Sense Varaibles
static nrf_saadc_value_t m_buffer[SAMPLES_IN_BUFFER];
float batF;
uint16_t bat16;
bool ADC_DONE;

// Time variables
uint32_t tot_sleep, bme_start, bme_wait,
    hdc2010_start, hdc2010_wait, gas_start, gas_wait, start_time, stop_time;
//i2c scan variables
enum {
  HDC2010_i2c,
  MAX44009_i2c,
  BME680_i2c,
  LAST_I2C
};
uint8_t i2c_device[LAST_I2C][2] = {
    {HDC2010_ADDR, false}, //HDC2010 65 in dec
    {MAX_ADDR, false},     //MAX44009 75 in dec
    {BME_ADDR, false}      //BME680 119 in dec
};

//BME680 Variables
bool GAS_ON = false;
uint8_t gas_counter = GAS_READ_RATIO - 2;
uint8_t bme_read_time = 12; //millis
uint8_t gas_read_time = GAS_HEAT_DURATION + 1;

//HDC2010 Variables
uint16_t hdc2010_read_time = 600; //micros
uint16_t hdc2010_hum16;
int16_t hdc2010_temp16;
float hdc2010_tempF, hdc2010_humF;
uint8_t hdc2010_data[4];
uint8_t hdc2010_temp_data[2], hdc2010_hum_data[2];
static double pow16;

//MAX variables
uint8_t max_lux[2];
uint8_t max_lux_exponent;
uint8_t max_reg = 0x03;
uint32_t max_light;
uint16_t max_light16;

//Radio Variables
enum radio_mode {
  CODED,
  Mbit_1,
  Mbit_2
};
enum data_type {
  WAKE_TYPE,
  DATA_TYPE,
  ACK_TYPE
};
uint8_t powerLVL[11] = {
    0xD8UL, // -40 dbm
    0xECUL, // -20 dbm
    0xF4UL, // -12 dbm
    0xFCUL, // -4 dbm
    0x0UL,  // 0 dbm
    0x2UL,  // 2 dbm
    0x4UL,  // 3 dbm
    0x5UL,  // 5 dbm
    0x6UL,  // 6 dbm
    0x7UL,  // 7 dbm
    0x8UL   // 8 dbm
};
uint8_t currentPL = START_PA;
bool channel_idle = false;
uint16_t packetNR = 0;
uint8_t radio_mem[20];
uint8_t dataArray[25];
uint8_t RSSI;
uint8_t pre0[4], pre1[4];
uint32_t pre_addr0, pre_addr1, base0, base1;
uint64_t ack_timer;
bool ack;
uint8_t ack_num;

APP_TIMER_DEF(wakeup_timer);
APP_TIMER_DEF(sleep_timer);

//Function declarations
static void wakeup(void *p_context);
void radio_setup(uint8_t mode);
void start_HFCLK();
void delay_sleep_ms(uint32_t sleep_time);
void start_listening();
void stop_radio();
void send_data();
void wait_for_ack();
void flush_radio_mem() {
  for (int i = 0; i < sizeof(radio_mem); i++) {
    radio_mem[i] = 0;
  }
}
uint32_t millis() {
  return (app_timer_cnt_get() / 32.768);
}
uint32_t micros() {
  return (app_timer_cnt_get() / 0.032786);
}
void saadc_callback(nrf_drv_saadc_evt_t const *p_event) {
  if (p_event->type == NRF_DRV_SAADC_EVT_DONE) {
    ret_code_t err_code;

    err_code = nrf_drv_saadc_buffer_convert(p_event->data.done.p_buffer,
          SAMPLES_IN_BUFFER);
    APP_ERROR_CHECK(err_code);
    int i;
    batF = 0;
    for (i = 0; i < SAMPLES_IN_BUFFER; i++) {
      batF = batF + p_event->data.done.p_buffer[i];
    }
    batF = batF * ADC_FACTOR;
    bat16 = batF;
    NRF_LOG_INFO("battery voltage is %d", batF);
    nrf_gpio_pin_clear(ENABLE_PIN);
    ADC_DONE = true;
  }
}
void saadc_init(void) {

  ret_code_t err_code;
  nrf_saadc_channel_config_t channel_config =
      NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN4);
  channel_config.gain = SAADC_CH_CONFIG_GAIN_Gain1;

  err_code = nrf_drv_saadc_init(NULL, saadc_callback);
  APP_ERROR_CHECK(err_code);

  err_code = nrf_drv_saadc_channel_init(0, &channel_config);
  APP_ERROR_CHECK(err_code);

  err_code = nrf_drv_saadc_buffer_convert(m_buffer, SAMPLES_IN_BUFFER);
  APP_ERROR_CHECK(err_code);
}
void i2c_init(void) {
  nrf_drv_twi_config_t twi_config;

  twi_config.sda = ARDUINO_SDA_PIN;
  twi_config.scl = ARDUINO_SCL_PIN;
  twi_config.frequency = NRF_TWI_FREQ_100K;
  twi_config.interrupt_priority = APP_IRQ_PRIORITY_HIGH;
  twi_config.clear_bus_init = false;

  nrf_drv_twi_init(&m_twi, &twi_config, NULL, NULL);
  nrf_drv_twi_enable(&m_twi);
}
...
```

B.1.1 I^2C scanner

Code B.2: I^2C scanner

```
...
void i2c_scanner(void) {
  ret_code_t err_code;
  uint8_t sample_data;
  NRF_LOG_INFO("Scanning for i2c devices...\n");
  for (uint8_t address = 0; address <= 127; address++) {
    err_code = nrf_drv_twi_rx(&m_twi, address, &sample_data, sizeof(sample_data));
    if (err_code == NRF_SUCCESS) {
      NRF_LOG_INFO("TWI device detected at address 0x%x.", address);
      for (uint8_t i = 0; i < LAST_I2C; i++) {
        if (address == i2c_device[i][0]) {
          i2c_device[i][1] = true;
        }
      }
    }
  }
  NRF_LOG_INFO("\n");
}
...
```

B.1.2 HDC2010

Code B.3: HDC2010

```
...
void hdc2010_init(void) {
  uint8_t hdc2010_reg_data[2] = {0x0e, 0x00};
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_reg_data, 2, false);
}
void hdc2010_heater(void) {
  uint8_t hdc2010_heater_data[2] = {0x0e, 0x08};
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_heater_data, 2, false);
  hdc2010_heater_data[1] = 0x00;
  delay_sleep_ms(10);
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_heater_data, 2, false);
}
void hdc2010_ask_data(void) {
  uint8_t hdc2010_ask_data[2] = {0x0f, 0xa1}; //0xa1
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_ask_data, 2, false);
}
void hdc2010_read_temp(void) {
  pow16 = pow(2, 16);
  uint8_t hdc2010_data_reg = 0x00;
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, &hdc2010_data_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, HDC2010_ADDR, hdc2010_temp_data, sizeof(hdc2010_temp_data));
  hdc2010_temp16 = hdc2010_temp_data[1] << 8 | hdc2010_temp_data[0];
  hdc2010_temp16 = (hdc2010_temp16 / pow16) * 165 - 40;
}
void hdc2010_read_hum(void) {
  pow16 = pow(2, 16);
  uint8_t hdc2010_data_reg = 0x02;
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, &hdc2010_data_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, HDC2010_ADDR, hdc2010_hum_data, sizeof(hdc2010_hum_data));
  hdc2010_hum16 = hdc2010_hum_data[1] << 8 | hdc2010_hum_data[0];
  hdc2010_hum16 = (hdc2010_hum16 / pow16) * 100;
}
void hdc2010_read(void) {
  pow16 = pow(2, 16);
  uint8_t hdc2010_data_reg = 0x00;
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, &hdc2010_data_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, HDC2010_ADDR, hdc2010_data, sizeof(hdc2010_data));
  hdc2010_tempF = hdc2010_data[1] << 8 | hdc2010_data[0];
  hdc2010_tempF = (hdc2010_tempF / pow16) * 165 - 40;
  hdc2010_humF = hdc2010_data[3] << 8 | hdc2010_data[2];
  hdc2010_humF = (hdc2010_humF / pow16) * 100;
  hdc2010_temp16 = hdc2010_tempF * 100;
  hdc2010_hum16 = hdc2010_humF * 100;
}
...
```

B.1.3 MAX44009

Code B.4: MAX44009

```
...
uint32_t max_read(void) {
  nrf_drv_twi_tx(&m_twi, MAX_ADDR, &max_reg, 1, false);
  nrf_drv_twi_rx(&m_twi, MAX_ADDR, max_lux, sizeof(max_lux));

  max_lux_exponent = ((max_lux[0] >> 4) & 0x0F);
  max_lux[0] = ((max_lux[0] << 4) & 0xF0);
  max_lux[1] &= 0x0F;

  max_light = 45L * (max_lux[0] | max_lux[1]) * (1 << max_lux_exponent);
  max_light = max_light / 1000;
  return max_light;
}
...
```

```
...
void send_packet() {
  // send the packet:
  NRF_RADIO->TASKS_RXEN = 0;
  NRF_RADIO->EVENTS_READY = 0U;
  NRF_RADIO->TASKS_TXEN = 1;

  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->EVENTS_END = 0U;
  NRF_RADIO->TASKS_START = 1U;

  while (NRF_RADIO->EVENTS_END == 0U) {
    // wait
  }

  stop_radio();
}
void stop_radio() {
  NRF_RADIO->EVENTS_DISABLED = 0U;
  // Disable radio
  NRF_RADIO->TASKS_DISABLE = 1U;

  while (NRF_RADIO->EVENTS_DISABLED == 0U) {
    // wait
  }
}
void start_listening() {
  //Start Listening
  NRF_RADIO->EVENTS_READY = 0U;
  // Enable radio and wait for ready
  NRF_RADIO->TASKS_RXEN = 1U;

  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->EVENTS_END = 0U;
  // Start listening and wait for address received event
  NRF_RADIO->TASKS_START = 1U;
}
static void power_manage(void) {
  nrf_drv_saadc_uninit();
  NRF_CLOCK->TASKS_HFCLKSTOP = 1;
  NVIC_ClearPendingIRQ(SAADC_IRQn);
  __set_FPSCR(__get_FPSCR() & ~(0x0000009F));
  (void)__get_FPSCR();
  NVIC_ClearPendingIRQ(FPU_IRQn);
  __SEV();
  __WFE();
  __WFE();
}
void sleep_wake(void *p_context) {
  UNUSED_PARAMETER(p_context);
  start_HFCLK();
}
void delay_sleep_ms(uint32_t sleep_time) {
  if (sleep_time > 20) {
    tot_sleep = tot_sleep + (sleep_time - 1);
    app_timer_start(sleep_timer, sleep_time - 2, NULL);
    NRF_CLOCK->TASKS_HFCLKSTOP = 1;
  } else {
    nrf_delay_ms(sleep_time);
  }
}
void start_LFCLK() {
  /* Start low frequency crystal oscillator for app_timer(used by bsp)*/
  NRF_CLOCK->LFCLKSRC = (CLOCK_LFCLKSRC_SRC_Xtal << CLOCK_LFCLKSRC_SRC_Pos);
  //NRF_CLOCK->LFRCMODE = 1;
  NRF_CLOCK->EVENTS_LFCLKSTARTED = 0;
  NRF_CLOCK->TASKS_LFCLKSTART = 1;

  while (NRF_CLOCK->EVENTS_LFCLKSTARTED == 0) {
    // Do nothing.
  }
}
void start_HFCLK(void) {
  NRF_CLOCK->EVENTS_HFCLKSTARTED = 0;
  NRF_CLOCK->TASKS_HFCLKSTART = 1;
  /* Wait for the external oscillator to start up */
  while (NRF_CLOCK->EVENTS_HFCLKSTARTED == 0) {
    // Do nothing.
  }
}
...
```

B.1.4 get_sensor_data()

Code B.5

```
...
static void get_sensor_data(void) {
  packetNR++;
  gas_counter++;
  nrf_gpio_pin_set(ENABLE_PIN);
  ADC_DONE = false;
  nrf_drv_saadc_sample();
  i2c_init();
  if (i2c_device[BME680_i2c][1]) {
    if (gas_counter >= GAS_READ_RATIO) {
      gas_counter = 0;
      GAS_ON = true;
      bme680_gas_on();
    }
    bme680_ask_data();
    bme_start = millis();
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_ask_data();
    hdc2010_start = micros();
  }
  if (i2c_device[MAX44009_i2c][1]) {
    max_light = max_read();
    max_light16 = max_light;
  }
  if (i2c_device[HDC2010_i2c][1]) {
    while ((hdc2010_start + hdc2010_read_time) > micros()) {
      // wait for hdc2010 data
    }
    hdc2010_read();
  }
  if (i2c_device[BME680_i2c][1]) {
    while ((bme_start + bme_read_time) > millis()) {
      // wait for bme680 data
    }
    bme680_get_data();
    if (GAS_ON == true) {
      while ((bme_start + bme_read_time + gas_read_time) > millis()) {
        // wait for bme680 data
      }
      GAS_ON = false;
      bmeGas = bme680_gas_read();
      bmeGas = bmeGas / 10000;
      bmeGas16 = bmeGas;
      bme680_gas_off();
    }
    humD = bme680_hum_calc() * 100;
    bmeHum16 = humD;
    bmePres16 = presD = bme680_pres_calc();
    tempD = bme680_temp_calc() * 100;
    bmeTemp16 = tempD;
  }
  nrf_drv_twi_disable(&m_twi);

  uint8_t byte = 2;
  radio_mem[byte++] = DATA_TYPE; // Data type;
  radio_mem[byte++] = 0;         // destID;
  radio_mem[byte++] = NODE;     // myID;
  radio_mem[byte++] = NODE;     // sourceID;
  radio_mem[byte++] = currentPL; // currentPL
  radio_mem[byte++] = (uint8_t)(packetNR >> 8);
  radio_mem[byte++] = (uint8_t)(packetNR);
  radio_mem[byte++] = (uint8_t)(bat16 >> 8);
  radio_mem[byte++] = (uint8_t)(bat16);

  if (i2c_device[MAX44009_i2c][1]) {
    radio_mem[byte++] = (uint8_t)(max_light16 >> 8);
    radio_mem[byte++] = (uint8_t)(max_light16);
    NRF_LOG_INFO("\n\rmax_light = %d\n",
        max_light16);
  }
  if (i2c_device[BME680_i2c][1]) {
    radio_mem[byte++] = (uint8_t)(bmeTemp16 >> 8);
    radio_mem[byte++] = (uint8_t)(bmeTemp16);
    radio_mem[byte++] = (uint8_t)(bmeHum16 >> 8);
    radio_mem[byte++] = (uint8_t)(bmeHum16);
    radio_mem[byte++] = (uint8_t)(bmePres16 >> 8);
    radio_mem[byte++] = (uint8_t)(bmePres16);
    radio_mem[byte++] = (uint8_t)(bmeGas16 >> 8);
    radio_mem[byte++] = (uint8_t)(bmeGas16);

    NRF_LOG_INFO("\n\rBME680:\n\rTemp = %d\n\rHum = %d\n\rPres = %d\n\rGas = %d\n\r",
        bmeTemp16,
        bmeHum16,
        bmePres16,
        bmeGas16);
  }
  if (i2c_device[HDC2010_i2c][1]) {
    radio_mem[byte++] = (uint8_t)(hdc2010_temp16 >> 8);
    radio_mem[byte++] = (uint8_t)(hdc2010_temp16);
    radio_mem[byte++] = (uint8_t)(hdc2010_hum16 >> 8);
    radio_mem[byte++] = (uint8_t)(hdc2010_hum16);
    NRF_LOG_INFO("\n\rHDC2010:\n\rTemp = %d\n\rHum = %d\n",
        hdc2010_temp16,
        hdc2010_hum16);
  }
  radio_mem[0] = byte - 2; // NR of payload bytes
  radio_mem[1] = 0;        // Unused

  for (int i = 0; i < radio_mem[0] + 2; i++) {
    NRF_LOG_INFO("%d", radio_mem[i]);
  }
  while (ADC_DONE = false) {
    //wait for adc
  }
}
bool carrier_sense() {
  NRF_RADIO->TASKS_RXEN = 1U;
  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->TASKS_EDSTART = 1;
  nrf_delay_us(130);
  if (NRF_RADIO->EDSAMPLE < 1) {
    return 1;
  } else {
    return 0;
  }
}
...
```

B.1.5 send_data()

Code B.6: send_data()

```
...
void send_data() {
  ack = false;
  ack_num = 0;
  NRF_LOG_INFO("Sending packet!");
  while (ack == false) {
    radio_mem[6] = currentPL;
    while (carrier_sense() == false) {
    }
    send_packet();
    NRF_LOG_INFO("Waiting for ack...");
    wait_for_ack();
    if (ack_num >= 4) {
      NRF_LOG_INFO("No ack received!!!!");
      break;
    }
  }
  NRF_LOG_INFO("Current Power level: %d", currentPL);
}
void wait_for_ack() {
  start_listening();
  ack_num++;
  ack_timer = micros() + 1500;
  while (NRF_RADIO->EVENTS_END == 0U) {
    // wait
    if (ack_timer < micros()) {
      break;
    }
  }
  if (NRF_RADIO->CRCSTATUS == 1U) {
    if ((radio_mem[2] == ACK_TYPE) && (radio_mem[3] == NODE)) {
      RSSI = NRF_RADIO->RSSISAMPLE;
      NRF_LOG_INFO("RSSI = -%d", RSSI);
      NRF_LOG_INFO("Received ack after %d microsec", micros() - (ack_timer - 1500));
      ack = true;
      stop_radio();
      for (int i = 0; i < radio_mem[0] + 2; i++) {
        NRF_LOG_INFO("%d", radio_mem[i]);
      }
      if (RSSI < radio_mem[4]) // Check the lowest RSSI
        RSSI = radio_mem[4];
      if ((RSSI < RSSI_UPPER_LIMIT) && (currentPL > 0)) {
        currentPL--;
        NRF_RADIO->TXPOWER = powerLVL[START_PA]; //currentPL
        NRF_LOG_INFO("Current Power level changed to: %d", currentPL);
      } else if ((RSSI > RSSI_LOWER_LIMIT) && (currentPL < 10)) {
        currentPL++;
        NRF_RADIO->TXPOWER = powerLVL[START_PA];
        NRF_LOG_INFO("Current Power level changed to: %d", currentPL);
      }
    }
    stop_radio();
    //flush_radio_mem();
    start_listening();
  }
  stop_radio();
  if ((ack == false) && (currentPL < 10)) {
    currentPL++;
    NRF_RADIO->TXPOWER = powerLVL[START_PA];
  }
}
...
```

```
static void wakeup(void *p_context) {
  UNUSED_PARAMETER(p_context);

  start_HFCLK();
  start_time = millis();
  saadc_init();
  get_sensor_data();
  send_data();
  NRF_LOG_INFO("\n\rOn-time = %d\n\n", (millis() - start_time));
}
...
```

B.1.6 Radio

Code B.7: radio_setup()

```
...
void radio_setup(uint8_t mode) {

  if (mode == CODED) {
    NRF_LOG_INFO("Using Coded!");
    uint32_t preamble_mask = (RADIO_PCNF0_PLEN_LongRange << RADIO_PCNF0_PLEN_Pos) |
                             (2 << RADIO_PCNF0_CILEN_Pos) |
                             (3 << RADIO_PCNF0_TERMLEN_Pos);

    NRF_RADIO->PCNF0 = (1UL << RADIO_PCNF0_S1LEN_Pos) |
                       (2UL << RADIO_PCNF0_S0LEN_Pos) |
                       (8UL << RADIO_PCNF0_LFLEN_Pos) |
                       preamble_mask;

    NRF_RADIO->PCNF1 = (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
                       (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos) |
                       (3UL << RADIO_PCNF1_BALEN_Pos) |
                       (0 << RADIO_PCNF1_STATLEN_Pos) |
                       (250 << RADIO_PCNF1_MAXLEN_Pos);
  } else {
    NRF_RADIO->PCNF0 = (0 << RADIO_PCNF0_S1LEN_Pos) |
                       (0 << RADIO_PCNF0_S0LEN_Pos) |
                       (8 << RADIO_PCNF0_LFLEN_Pos);

    NRF_RADIO->PCNF1 = (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
                       (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos) |
                       (4 << RADIO_PCNF1_BALEN_Pos) |
                       (0 << RADIO_PCNF1_STATLEN_Pos) |
                       (200 << RADIO_PCNF1_MAXLEN_Pos);
  }
  NRF_RADIO->CRCCNF = (RADIO_CRCCNF_LEN_Two << RADIO_CRCCNF_LEN_Pos); // checksum bits
  if ((NRF_RADIO->CRCCNF & RADIO_CRCCNF_LEN_Msk) ==
      (RADIO_CRCCNF_LEN_Two << RADIO_CRCCNF_LEN_Pos)) {
    NRF_RADIO->CRCINIT = 0xFFFFUL;  // Initial value
    NRF_RADIO->CRCPOLY = 0x11021UL; // CRC poly: x^16 + x^12^x^5 + 1
  } else if ((NRF_RADIO->CRCCNF & RADIO_CRCCNF_LEN_Msk) ==
             (RADIO_CRCCNF_LEN_One << RADIO_CRCCNF_LEN_Pos)) {
    NRF_RADIO->CRCINIT = 0xFFUL;  // Initial value
    NRF_RADIO->CRCPOLY = 0x107UL; // CRC poly: x^8 + x^2^x^1 + 1
  }

  if (mode == CODED) {
    NRF_RADIO->CRCCNF = (RADIO_CRCCNF_SKIPADDR_Skip << RADIO_CRCCNF_SKIPADDR_Pos) |
                        (RADIO_CRCCNF_LEN_Three << RADIO_CRCCNF_LEN_Pos);
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Ble_LR125Kbit << RADIO_MODE_MODE_Pos);
  } else if (mode == Mbit_1) {
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Nrf_1Mbit << RADIO_MODE_MODE_Pos);
  } else if (mode == Mbit_2) {
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Nrf_2Mbit << RADIO_MODE_MODE_Pos);
  }

  pre0[0] = 0x02;
  pre0[1] = 0x11;
  pre0[2] = 0x22;
  pre0[3] = 0x01;

  pre1[0] = 0x02;
  pre1[1] = 0x16;
  pre1[2] = 0x2a;
  pre1[3] = 0x3e;

  pre_addr0 = ((pre0[3] << 24) | (pre0[2] << 16) | (pre0[1] << 8) | (pre0[0]));
  pre_addr1 = ((pre1[3] << 24) | (pre1[2] << 16) | (pre1[1] << 8) | (pre1[0]));
  base0 = 0x11111111;
  base1 = 0x22222222;
  NRF_RADIO->BASE0 = (uint32_t)(base0);
  NRF_RADIO->BASE1 = (uint32_t)(base1);
  NRF_RADIO->PREFIX0 = (uint32_t)(pre_addr0);
  NRF_RADIO->PREFIX1 = (uint32_t)(pre_addr1);
  NRF_RADIO->TXADDRESS = 0;
  NRF_RADIO->RXADDRESSES = 0b00000001;
  NRF_RADIO->PACKETPTR = (uint32_t)&radio_mem;
  NRF_RADIO->SHORTS |= RADIO_SHORTS_ADDRESS_RSSISTART_Msk;
  NRF_RADIO->FREQUENCY = 7UL;
  NRF_RADIO->TXPOWER = powerLVL[currentPL];
}
```

```
void turn_ram_off(uint8_t RAM) {
  if (RAM < 2) { // dont turn off first 2 RAM blocks
  } else {
    NRF_POWER->RAM[RAM].POWERSET = 0b0000000000000000;
  }
}
int main(void) {
  //Setup
  start_HFCLK();
  start_LFCLK();
  NRF_LOG_INIT(NULL);
  NRF_LOG_DEFAULT_BACKENDS_INIT();
  NRF_LOG_INFO("\n\r################Star Node Start!#################\n");
  app_timer_init();
  app_timer_create(&wakeup_timer, APP_TIMER_MODE_REPEATED, wakeup);
  app_timer_create(&sleep_timer, APP_TIMER_MODE_SINGLE_SHOT, sleep_wake);
  app_timer_start(wakeup_timer, SEND_INTERVAL, NULL);
  radio_setup(Mbit_1);
  i2c_init();
  i2c_scanner();
  if (i2c_device[BME680_i2c][1]) {
    bme680_calib();
    bme680_reset();
    bme680_iir_filter();
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_init();
  }
  nrf_drv_twi_disable(&m_twi);
  nrf_gpio_cfg_output(ENABLE_PIN);
  //for (int i = 3; i <= 8; i++) {
  //  turn_ram_off(i);
  //}
  for (;;) {
    if (NRF_LOG_PROCESS() == false) {
      power_manage();
    }
  }
}
```

B.2 Multi-hop

Code B.8: Multi-hop Code

```
/*##########################################
  ##### Multi-hop Node #####################
  ##########################################*/

#define NRF_LOG_ENABLED 1 // enable/disable prints
#define START_HOPS 100
#define NODE 115
#define START_PA_LEVEL 10 // Initial power level
#define BROADCAST_ADDR 0xff
#define GAS_READ_RATIO 10
#define WAKE_INTERVAL 1 * 2000 // node wake_ping interval
#define SEND_INTERVAL 3        // Intervals of wake_ping before sending sensor data
#define ENABLE_PIN NRF_GPIO_PIN_MAP(1, 8)
#define HDC2010_ADDR (0x41)
```

```
#define MAX_ADDR (0x4b)
#define SAMPLES_IN_BUFFER 1
#define ADC_FACTOR 5390.625 // Precalculated ADC factor
#define WAKEPING_INTERVAL APP_TIMER_TICKS(WAKE_INTERVAL)

#include "app_error.h"
#include "app_timer.h"
#include "app_uart.h"
#include "boards.h"
#include "bsp.h"
#include "nordic_common.h"
#include "nrf_bme680.h"
#include "nrf_delay.h"
#include "nrf_drv_clock.h"
#include "nrf_drv_saadc.h"
#include "nrf_drv_twi.h"
#include "nrf_error.h"
#include "nrf_gpio.h"
#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

//Radio Variables
enum radio_mode {
  CODED,
  Mbit_1,
  Mbit_2
};
enum data_type {
  WAKE_TYPE,
  DATA_TYPE,
  ACK_TYPE
};
uint8_t powerLVL[11] = {
    0xD8UL, // -40 dbm
    0xECUL, // -20 dbm
    0xF4UL, // -12 dbm
    0xFCUL, // -4 dbm
    0x0UL,  // 0 dbm
    0x2UL,  // 2 dbm
    0x4UL,  // 3 dbm
    0x5UL,  // 5 dbm
    0x6UL,  // 6 dbm
    0x7UL,  // 7 dbm
    0x8UL   // 8 dbm
};
uint8_t currentPL = START_PA_LEVEL;
uint8_t my_hops_to_sink = START_HOPS;
uint16_t packetNR = 0;
uint8_t radio_mem[20];
uint8_t wake_mem[10];
uint8_t data_mem[25];
```

```
uint8_t ack_mem[10];
uint8_t dataArray[25];
uint8_t pre0[4], pre1[4];
uint32_t pre_addr0, pre_addr1, base0, base1;
uint64_t ack_timer;
bool timeout;
uint8_t ack_num;
uint8_t send_counter;
uint32_t timeout_time, current_time, relay_time, start_time;
uint8_t next_hop;
uint8_t temp_hops;

//BME680 Variables
bool GAS_ON = false;
uint8_t gas_counter = GAS_READ_RATIO - 8;
uint8_t bme_read_time = 12;
uint16_t hdc2010_read_time = 600;
uint8_t gas_read_time = GAS_HEAT_DURATION + 1;

// i2c scan variables
enum {
  HDC2010_i2c,
  MAX44009_i2c,
  BME680_i2c,
  LAST_I2C
};
uint8_t i2c_device[LAST_I2C][2] = {
    {HDC2010_ADDR, false}, //HDC2010 65 in dec
    {MAX_ADDR, false},     //MAX44009 75 in dec
    {BME_ADDR, false}      //BME680 119 in decl
};

//Battery Sense Varaibles
static nrf_saadc_value_t m_buffer[SAMPLES_IN_BUFFER];
float batF;
uint16_t bat16;
uint8_t bat8;
bool ADC_DONE;

//HDC2010 Variables
uint16_t hdc2010_temp16, hdc2010_hum16;
uint8_t hdc2010_data[4];
uint8_t hdc2010_temp_data[2], hdc2010_hum_data[2];
static double pow16;

//MAX variables
uint8_t max_lux[2];
uint8_t max_lux_exponent;
uint8_t max_reg = 0x03;
uint32_t max_light;
uint16_t max_light16;

// Timers
uint32_t tot_sleep, tsl_start, tsl_wait, bme_start, bme_wait,
    hdc2010_start, hdc2010_wait, stop_time, gas_start, gas_wait;

uint8_t teller = 0;
```

```

APP_TIMER_DEF(m_wakeping_timer);
APP_TIMER_DEF(wakeping_timer);
APP_TIMER_DEF(sleep_timer);

bool relay_data();
void change_next_hop_addr(uint8_t new_next_hop);
void send_ack();
bool wait_for_ack();
bool carrier_sense();
void print_received_data();
void wait_to_send();
void send_packet();
void send_data();
void wake_ping();
void get_sensor_data();
static void wakeup(void *p_context);
void start_listening();
void stop_radio();
bool wait_for_response();
void delay_sleep_ms(uint32_t sleep_time);

void uart_error_handle(app_uart_evt_t *p_event) {
  if (p_event->evt_type == APP_UART_COMMUNICATION_ERROR) {
    APP_ERROR_HANDLER(p_event->data.error_communication);
  } else if (p_event->evt_type == APP_UART_FIFO_ERROR) {
    APP_ERROR_HANDLER(p_event->data.error_code);
  }
}
void flush_radio_mem() {
  for (int i = 0; i < sizeof(radio_mem); i++) {
    radio_mem[i] = 0;
  }
}
void flush_wake_mem() {
  for (int i = 0; i < sizeof(wake_mem); i++) {
    wake_mem[i] = 0;
  }
}
void flush_data_mem() {
  for (int i = 0; i < sizeof(data_mem); i++) {
    data_mem[i] = 0;
  }
}
void flush_ack_mem() {
  for (int i = 0; i < sizeof(ack_mem); i++) {
    ack_mem[i] = 0;
  }
}
uint32_t millis() {
  return (app_timer_cnt_get() / 32.768);
}
uint32_t micros() {
  return (app_timer_cnt_get() / 0.032786);
}
void saadc_callback(nrf_drv_saadc_evt_t const *p_event) {
  if (p_event->type == NRF_DRV_SAADC_EVT_DONE) {
    ret_code_t err_code;

    err_code = nrf_drv_saadc_buffer_convert(p_event->data.done.p_buffer,
        SAMPLES_IN_BUFFER);
    APP_ERROR_CHECK(err_code);
    int i;
    batF = 0;
    for (i = 0; i < SAMPLES_IN_BUFFER; i++) {
      batF = batF + p_event->data.done.p_buffer[i];
    }
    batF = batF * ADC_FACTOR;
    bat16 = batF;
    nrf_gpio_pin_clear(ENABLE_PIN);
    ADC_DONE = true;
  }
}

void saadc_init(void) {

  ret_code_t err_code;
  nrf_saadc_channel_config_t channel_config =
      NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN4);
  channel_config.gain = SAADC_CH_CONFIG_GAIN_Gain1;
  err_code = nrf_drv_saadc_init(NULL, saadc_callback);
  APP_ERROR_CHECK(err_code);

  err_code = nrf_drv_saadc_channel_init(0, &channel_config);
  APP_ERROR_CHECK(err_code);

  err_code = nrf_drv_saadc_buffer_convert(m_buffer, SAMPLES_IN_BUFFER);
  APP_ERROR_CHECK(err_code);
}
void i2c_init(void) {
  nrf_drv_twi_config_t twi_config;

  twi_config.sda = ARDUINO_SDA_PIN;
  twi_config.scl = ARDUINO_SCL_PIN;
  twi_config.frequency = NRF_TWI_FREQ_100K;
  twi_config.interrupt_priority = APP_IRQ_PRIORITY_HIGH;
  twi_config.clear_bus_init = false;

  nrf_drv_twi_init(&m_twi, &twi_config, NULL, NULL);
  nrf_drv_twi_enable(&m_twi);
}
void i2c_scanner(void) {
  ret_code_t err_code;
  uint8_t sample_data;
  NRF_LOG_INFO("Scanning for i2c devices...\n");
  for (uint8_t address = 0; address <= 127; address++) {
    err_code = nrf_drv_twi_rx(&m_twi, address, &sample_data, sizeof(sample_data));
    nrf_delay_us(5);
    if (err_code == NRF_SUCCESS) {
      NRF_LOG_INFO("TWI device detected at address 0x%x.", address);
      for (uint8_t i = 0; i < LAST_I2C; i++) {
        if (address == i2c_device[i][0]) {
          i2c_device[i][1] = true;
        }
      }
    }
  }
  NRF_LOG_INFO("\n");
}
void hdc2010_init(void) {
  uint8_t hdc2010_reg_data[2] = {0x0e, 0x00};
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_reg_data, 2, false);
}
void hdc2010_heater(void) {
  uint8_t hdc2010_heater_data[2] = {0x0e, 0x08};
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_heater_data, 2, false);
  hdc2010_heater_data[1] = 0x00;
  delay_sleep_ms(10);
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_heater_data, 2, false);
}
void hdc2010_ask_data(void) {
  uint8_t hdc2010_ask_data[2] = {0x0f, 0xa1}; //0xa1
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, hdc2010_ask_data, 2, false);
}
void hdc2010_read_temp(void) {
  pow16 = pow(2, 16);
  uint8_t hdc2010_data_reg = 0x00;
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, &hdc2010_data_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, HDC2010_ADDR, hdc2010_temp_data, sizeof(hdc2010_temp_data));
  hdc2010_temp16 = hdc2010_temp_data[1] << 8 | hdc2010_temp_data[0];
  hdc2010_temp16 = (hdc2010_temp16 / pow16) * 165 - 40;
}
void hdc2010_read_hum(void) {
  pow16 = pow(2, 16);
  uint8_t hdc2010_data_reg = 0x02;
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, &hdc2010_data_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, HDC2010_ADDR, hdc2010_hum_data, sizeof(hdc2010_hum_data));
  hdc2010_hum16 = hdc2010_hum_data[1] << 8 | hdc2010_hum_data[0];
  hdc2010_hum16 = (hdc2010_hum16 / pow16) * 100;
}
void hdc2010_read(void) {
  pow16 = pow(2, 16);
  uint8_t hdc2010_data_reg = 0x00;
  nrf_drv_twi_tx(&m_twi, HDC2010_ADDR, &hdc2010_data_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, HDC2010_ADDR, hdc2010_data, sizeof(hdc2010_data));
  hdc2010_temp16 = hdc2010_data[1] << 8 | hdc2010_data[0];
  hdc2010_temp16 = (hdc2010_temp16 / pow16) * 165 - 40;
  hdc2010_hum16 = hdc2010_data[3] << 8 | hdc2010_data[2];
  hdc2010_hum16 = (hdc2010_hum16 / pow16) * 100;
}
uint32_t max_read(void) {
  nrf_drv_twi_tx(&m_twi, MAX_ADDR, &max_reg, 1, false);
  nrf_drv_twi_rx(&m_twi, MAX_ADDR, max_lux, sizeof(max_lux));

  max_lux_exponent = ((max_lux[0] >> 4) & 0x0F);
  max_lux[0] = ((max_lux[0] << 4) & 0xF0);
  max_lux[1] &= 0x0F;

  //lux_value    = 0.045 * ( lux_high + lux_low ) * (1<< lux_exponent);
  max_light = 45L * (max_lux[0] | max_lux[1]) * (1 << max_lux_exponent);
  max_light = max_light / 1000;
  return max_light;
}
void delay_sleep_ms(uint32_t sleep_time) {
  if (sleep_time > 50) {
    tot_sleep = tot_sleep + (sleep_time - 1);
    app_timer_start(sleep_timer, sleep_time - 2, NULL);
    NRF_CLOCK->TASKS_HFCLKSTOP = 1;
  } else {
    nrf_delay_ms(sleep_time);
  }
}

void print_radio_config() {
  NRF_LOG_INFO("\n\rPrefix0 = %u\n\rPrefix1 = %u\n\rBase0 =  %u\n\rBase1 = %u",
      NRF_RADIO->PREFIX0,
      NRF_RADIO->PREFIX1,
      NRF_RADIO->BASE0,
      NRF_RADIO->BASE1);
}
void print_msg() {
  NRF_LOG_INFO("Size: %d", radio_mem[0]);
  NRF_LOG_INFO("From: %d", radio_mem[2]);
  NRF_LOG_INFO("Hops: %d", radio_mem[3]);
  for (int i = 4; i < radio_mem[0]; i++) {
    NRF_LOG_INFO("data[%d] = %d", i - 4, radio_mem[i]);
  }
  NRF_LOG_INFO("");
}
void get_sensor_data() {
  //NRF_LOG_INFO("get_sensor_data");
  saadc_init();
  packetNR++;
  gas_counter++;
  nrf_gpio_pin_set(ENABLE_PIN);
  ADC_DONE = false;
  nrf_drv_saadc_sample();
  i2c_init();
  if (i2c_device[BME680_i2c][1]) {
    if (gas_counter == GAS_READ_RATIO - 1) {
      bme680_gas_on();
    } else if (gas_counter >= GAS_READ_RATIO) {
      gas_counter = 0;
      GAS_ON = true;
    }
    bme680_ask_data();
    bme_start = millis(); //milliseconds
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_ask_data();
    hdc2010_start = micros(); //microseconds
  }
  if (i2c_device[MAX44009_i2c][1]) {
    max_light = max_read();
    max_light16 = max_light;
  }
  if (i2c_device[HDC2010_i2c][1]) {
    while ((hdc2010_start + hdc2010_read_time) > micros()) {
      // wait for hdc2010 data
    }
    hdc2010_read();
  }
  if (i2c_device[BME680_i2c][1]) {
    //NRF_LOG_INFO("waiting bme");
    while ((bme_start + bme_read_time) > millis()) {
      // wait for bme680 data
    }
    bme680_get_data();
    if (GAS_ON == true) {
      GAS_ON = false;
      bmeGas = bme680_gas_read();
      bmeGas = bmeGas / 10000;
      bmeGas16 = bmeGas;
      bme680_gas_off();
    }
    humD = bme680_hum_calc() * 100;
    bmeHum16 = humD;
    bmePres16 = presD = bme680_pres_calc();
    tempD = bme680_temp_calc() * 100;
    bmeTemp16 = tempD;
  }
  nrf_drv_twi_disable(&m_twi);
  uint8_t byte = 2;
  dataArray[byte++] = DATA_TYPE;
  dataArray[byte++] = 0;    //destID;
  dataArray[byte++] = NODE; //myID;
  dataArray[byte++] = NODE; //sourceID;
  dataArray[byte++] = 0;    // hops
  dataArray[byte++] = (uint8_t)(packetNR >> 8);
  dataArray[byte++] = (uint8_t)(packetNR);
  dataArray[byte++] = (uint8_t)(bat16 >> 8);
  dataArray[byte++] = (uint8_t)(bat16);

  if (i2c_device[MAX44009_i2c][1]) {
    dataArray[byte++] = (uint8_t)(max_light16 >> 8);
    dataArray[byte++] = (uint8_t)(max_light16);
    //NRF_LOG_INFO("\n\rmax_light = %d\n",
    //  max_light16);
  }
  if (i2c_device[BME680_i2c][1]) {
    dataArray[byte++] = (uint8_t)(bmeTemp16 >> 8);
    dataArray[byte++] = (uint8_t)(bmeTemp16);
    dataArray[byte++] = (uint8_t)(bmeHum16 >> 8);
    dataArray[byte++] = (uint8_t)(bmeHum16);
    dataArray[byte++] = (uint8_t)(bmePres16 >> 8);
    dataArray[byte++] = (uint8_t)(bmePres16);
    dataArray[byte++] = (uint8_t)(bmeGas16 >> 8);
    dataArray[byte++] = (uint8_t)(bmeGas16);

    NRF_LOG_INFO("\n\rBME680:\n\rTemp = %d\n\rHum = %d\n\rPres = %d\n\rGas = %d\n\r",
        bmeTemp16,
        bmeHum16,
        bmePres16,
        bmeGas16);
  }
  if (i2c_device[HDC2010_i2c][1]) {
    dataArray[byte++] = (uint8_t)(hdc2010_temp16 >> 8);
    dataArray[byte++] = (uint8_t)(hdc2010_temp16);
    dataArray[byte++] = (uint8_t)(hdc2010_hum16 >> 8);
    dataArray[byte++] = (uint8_t)(hdc2010_hum16);
    // NRF_LOG_INFO("\n\rHDC2010:\n\rTemp = %d\n\rHum = %d\n",
    //   hdc2010_temp16,
    // hdc2010_hum16);
  }
  dataArray[0] = byte - 2;
  dataArray[1] = 0;

  for (int i = 0; i < dataArray[0] + 2; i++) {
    //NRF_LOG_INFO("%d", dataArray[i]);
  }

  while (ADC_DONE = false) {
    // waiting for adc
  }
  //NRF_LOG_INFO("adc done");
}
void send_packet() {

  // send the packet:
  NRF_RADIO->TASKS_RXEN = 0;

  NRF_RADIO->EVENTS_READY = 0U;
  NRF_RADIO->TASKS_TXEN = 1;

  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->EVENTS_END = 0U;
  NRF_RADIO->TASKS_START = 1U;

  while (NRF_RADIO->EVENTS_END == 0U) {
    // wait
  }
}
void stop_radio() {
  NRF_RADIO->EVENTS_DISABLED = 0U;
  // Disable radio
  NRF_RADIO->TASKS_DISABLE = 1U;

  while (NRF_RADIO->EVENTS_DISABLED == 0U) {
    // wait
  }
}
void start_listening() {
  //Start Listening
  NRF_RADIO->EVENTS_READY = 0U;
  // Enable radio and wait for ready
  NRF_RADIO->TASKS_RXEN = 1U;

  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->EVENTS_END = 0U;
  // Start listening and wait for address received event
  NRF_RADIO->TASKS_START = 1U;
}

void power_manage() {
  nrf_drv_saadc_uninit();
  NRF_CLOCK->TASKS_HFCLKSTOP = 1;
  NVIC_ClearPendingIRQ(SAADC_IRQn);
  __set_FPSCR(__get_FPSCR() & ~(0x0000009F));
  (void)__get_FPSCR();
  NVIC_ClearPendingIRQ(FPU_IRQn);

  __SEV();
  __WFE();
  __WFE();
}
void start_LFCLK() {
  /* Start low frequency crystal oscillator for app_timer(used by bsp)*/
  NRF_CLOCK->LFCLKSRC = (CLOCK_LFCLKSRC_SRC_Xtal << CLOCK_LFCLKSRC_SRC_Pos);
  //NRF_CLOCK->LFRCMODE = 1;
  NRF_CLOCK->EVENTS_LFCLKSTARTED = 0;
  NRF_CLOCK->TASKS_LFCLKSTART = 1;

  while (NRF_CLOCK->EVENTS_LFCLKSTARTED == 0) {
    // Do nothing.
  }
}
void start_HFCLK(void) {
  NRF_CLOCK->EVENTS_HFCLKSTARTED = 0;
  NRF_CLOCK->TASKS_HFCLKSTART = 1;
  /* Wait for the external oscillator to start up */
  while (NRF_CLOCK->EVENTS_HFCLKSTARTED == 0) {
    // Do nothing.
  }
}
void wake_ping() {
  //NRF_LOG_INFO("wake_ping");
  NRF_RADIO->TXPOWER = powerLVL[START_PA_LEVEL - 1];
  NRF_RADIO->TXADDRESS = 0;
  NRF_RADIO->PACKETPTR = (uint32_t)&wake_mem;
  uint8_t byte = 2;
  wake_mem[byte++] = WAKE_TYPE;
  wake_mem[byte++] = NODE;
  wake_mem[byte++] = my_hops_to_sink;
  wake_mem[0] = byte - 2;
  wake_mem[1] = 0;
  send_packet();
  flush_wake_mem();
  stop_radio();
  NRF_RADIO->TXPOWER = powerLVL[START_PA_LEVEL];
  //wait_for_response();
}
bool wait_for_response() {
  //NRF_LOG_INFO("wait_for_response");
  timeout_time = micros();
  timeout = false;
  while (1) {
    NRF_RADIO->RXADDRESSES = 0b00000100;
    NRF_RADIO->PACKETPTR = (uint32_t)&data_mem;
    NRF_RADIO->EVENTS_ADDRESS = 0;
    NRF_RADIO->EVENTS_END == 0;
    start_listening();
    while (!NRF_RADIO->EVENTS_ADDRESS) { // Wait for received address
      if (micros() > timeout_time + 1000) {
        timeout = true;
        NRF_LOG_INFO("timeout");
        stop_radio();
        flush_data_mem();
        return 0;
      }
    }
    if (timeout == false) {
      //NRF_LOG_INFO("Received address after %d micros", micros() - timeout_time);

      while (NRF_RADIO->EVENTS_END == 0U) { // wait for packet received
        if (micros() > timeout_time + 5000) {
          timeout = true;
          NRF_LOG_INFO("timeout");
          NRF_RADIO->EVENTS_ADDRESS = 0;
          NRF_RADIO->EVENTS_END = 0;
          stop_radio();
          flush_data_mem();
          return 0;
        }
      }
    }
    if ((NRF_RADIO->CRCSTATUS == 1U) && (NRF_RADIO->RXMATCH == 2)) {
      if ((data_mem[2] == DATA_TYPE) && (data_mem[3] == NODE)) {
        stop_radio();
        send_ack();
        //NRF_LOG_INFO("Relaying for %d after %d micros ", data_mem[4], micros() - timeout_time);
        data_mem[6]++;      // increase hops
        data_mem[4] = NODE; // change packet ID
        for (int i = 0; i < data_mem[0] + 2; i++) {
          dataArray[i] = data_mem[i];
        }
        return 1;
      }
    }
  }
}
void send_ack() {
  change_next_hop_addr(data_mem[4]);
  NRF_RADIO->PACKETPTR = (uint32_t)&ack_mem;
  NRF_RADIO->TXADDRESS = 1;
  uint8_t byte = 2;
  ack_mem[byte++] = ACK_TYPE;
  ack_mem[byte++] = data_mem[4]; // ID
  ack_mem[0] = byte - 2;
  ack_mem[1] = 0;
  nrf_delay_us(10);
  send_packet();
  flush_ack_mem();
  stop_radio();
}
bool send_to_gateway() {
  next_hop = 0x00;
  for (int i = 0; i < dataArray[0] + 2; i++) {
    data_mem[i] = dataArray[i];
  }
  data_mem[3] = next_hop;
  uint8_t retry = 5;
  while (1) {
    NRF_LOG_INFO("send_to_gateway");
    nrf_delay_us(900 + (2 * (rand() % 40)));
    NRF_RADIO->TXADDRESS = 1;
    NRF_RADIO->PACKETPTR = (uint32_t)&data_mem;
    change_next_hop_addr(next_hop);
    retry--;
    if (carrier_sense()) {
      send_packet();
      stop_radio();
      if (wait_for_ack()) {
        my_hops_to_sink = 1;
        flush_data_mem();
        return 1;
      } else {
        NRF_LOG_INFO("no ack received");
      }

    } else {
      NRF_LOG_INFO("Channel is busy");
    }
    if (retry <= 0) {
      flush_data_mem();
      return 0;
    }
  }
}
void wait_to_send() {
  //NRF_LOG_INFO("wait_to_send");
  if (my_hops_to_sink == 1) {
    send_to_gateway();
  }
  timeout_time = millis();
  while (my_hops_to_sink != 1) {
    NRF_RADIO->PACKETPTR = (uint32_t)&wake_mem;
    NRF_RADIO->RXADDRESSES = 0b00000001;
    start_listening();
    while (NRF_RADIO->EVENTS_END == 0U) {
      //wait
      if ((millis() > timeout_time + 2200) && (my_hops_to_sink < 253)) {
        NRF_LOG_INFO("timeout");
        //NRF_LOG_FLUSH();
        //my_hops_to_sink++;
        timeout_time = millis();
      }
    }
    NRF_RADIO->EVENTS_END = 0;
    stop_radio();
    if ((NRF_RADIO->CRCSTATUS == 1U) && (NRF_RADIO->RXMATCH == 0)) {
      if ((wake_mem[2] == WAKE_TYPE) && (wake_mem[4] < my_hops_to_sink)) {
        if (wake_mem[4] == 0) {
          if (send_to_gateway()) {
            break;
          }
        } else {
          temp_hops = wake_mem[4] + 1;
          //NRF_LOG_INFO("got wake ping from %d",wake_mem[3]);
          next_hop = wake_mem[3];
          for (int i = 0; i < dataArray[0] + 2; i++) {
            data_mem[i] = dataArray[i];
          }
          data_mem[3] = next_hop;
          nrf_delay_us(10 + (2 * (rand() % 40)));
          if (carrier_sense()) {
            change_next_hop_addr(next_hop);
            NRF_RADIO->PACKETPTR = (uint32_t)&data_mem;
            NRF_RADIO->TXADDRESS = 1;
            NRF_LOG_INFO("sending data");
            send_packet();
            stop_radio();
            if (wait_for_ack()) {
              my_hops_to_sink = temp_hops;
              NRF_LOG_INFO("ACK received, my hops = %d", my_hops_to_sink);
              break;
            } else {
              NRF_LOG_INFO("no ACK received");
              //flush_data_mem();
            }

          } else {
            NRF_LOG_INFO("channel busy");
            //flush_data_mem();
          }
        }
      }
    }
    flush_wake_mem();
    flush_data_mem();
  }
}

bool wait_for_ack() {
  //NRF_LOG_INFO("wait_for_ack");
  timeout_time = micros();
  timeout = false;
  while (!timeout) {
    NRF_RADIO->RXADDRESSES = 0b00000100;
    NRF_RADIO->PACKETPTR = (uint32_t)&ack_mem;
    NRF_RADIO->EVENTS_ADDRESS = 0;
    NRF_RADIO->EVENTS_END == 0;
    start_listening();
    while (NRF_RADIO->EVENTS_ADDRESS == 0) { // Wait for received address
      if (micros() > timeout_time + 800) {   //800
        timeout = true;
        stop_radio();
        flush_ack_mem();
        return 0;
      }
    }
    if (timeout == false) {
      //NRF_LOG_INFO("Received address after %d micros", micros() - timeout_time);
      while (NRF_RADIO->EVENTS_END == 0U) { // wait for packet received
        if (micros() > timeout_time + 1500) {
          timeout = true;
          stop_radio();
          flush_ack_mem();
          return 0;
        }
      }
    }
    if ((NRF_RADIO->CRCSTATUS == 1U) && (NRF_RADIO->RXMATCH == 2)) {
      if ((ack_mem[2] == ACK_TYPE) && (ack_mem[3] == NODE)) {
        stop_radio();
        //NRF_LOG_INFO("Received ack after %d micros", micros() - timeout_time);
        flush_ack_mem();
        return 1;
      }
    }
  }
}
bool carrier_sense() {
  NRF_RADIO->RXADDRESSES = 0b00000111;
  NRF_RADIO->TASKS_RXEN = 1U;
  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->TASKS_EDSTART = 1;
  nrf_delay_us(130);
  if (NRF_RADIO->EDSAMPLE < 1) {
    return 1;
  } else {
    return 0;
  }
}

void wakeup(void *p_context) {
  UNUSED_PARAMETER(p_context);
  start_HFCLK();
  //NRF_LOG_INFO("wakeup!");
  if ((send_counter >= SEND_INTERVAL) || (my_hops_to_sink > 199)) {
    get_sensor_data();
    wait_to_send();
    while (relay_data()) {
    }
    send_counter = 0;
  } else {
    while (relay_data()) {
    }
  }
  app_timer_start(wakeping_timer, WAKEPING_INTERVAL, NULL);
  send_counter++;
}
```

```
bool relay_data() {
  wake_ping();
  if (wait_for_response()) {
    wait_to_send();
    //NRF_LOG_FLUSH();
    return 1;
  }
  return 0;
}
void radio_setup(uint8_t mode) {

  if (mode == CODED) {
    NRF_LOG_INFO("Using Coded!");
    uint32_t preamble_mask = (RADIO_PCNF0_PLEN_LongRange << RADIO_PCNF0_PLEN_Pos) |
                             (2 << RADIO_PCNF0_CILEN_Pos) |
                             (3 << RADIO_PCNF0_TERMLEN_Pos);

    NRF_RADIO->PCNF0 = (1UL << RADIO_PCNF0_S1LEN_Pos) |
                       (2UL << RADIO_PCNF0_S0LEN_Pos) |
                       (8UL << RADIO_PCNF0_LFLEN_Pos) |
                       preamble_mask;

    NRF_RADIO->PCNF1 = (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
                       (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos) |
                       (3UL << RADIO_PCNF1_BALEN_Pos) |
                       (0 << RADIO_PCNF1_STATLEN_Pos) |
                       (200 << RADIO_PCNF1_MAXLEN_Pos);
  } else {
    NRF_RADIO->PCNF0 = (0 << RADIO_PCNF0_S1LEN_Pos) |
                       (0 << RADIO_PCNF0_S0LEN_Pos) |
                       (0 << RADIO_PCNF0_LFLEN_Pos);

    NRF_RADIO->PCNF1 = (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
                       (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos) |
                       (4 << RADIO_PCNF1_BALEN_Pos) |
                       (5 << RADIO_PCNF1_STATLEN_Pos) |
                       (200 << RADIO_PCNF1_MAXLEN_Pos);
  }
  NRF_RADIO->CRCCNF = (RADIO_CRCCNF_LEN_Two << RADIO_CRCCNF_LEN_Pos); // Number of checksum bits
  if ((NRF_RADIO->CRCCNF & RADIO_CRCCNF_LEN_Msk) ==
      (RADIO_CRCCNF_LEN_Two << RADIO_CRCCNF_LEN_Pos)) {
    NRF_RADIO->CRCINIT = 0xFFFFUL;  // Initial value
    NRF_RADIO->CRCPOLY = 0x11021UL; // CRC poly: x^16 + x^12^x^5 + 1
  } else if ((NRF_RADIO->CRCCNF & RADIO_CRCCNF_LEN_Msk) ==
             (RADIO_CRCCNF_LEN_One << RADIO_CRCCNF_LEN_Pos)) {
    NRF_RADIO->CRCINIT = 0xFFUL;  // Initial value
    NRF_RADIO->CRCPOLY = 0x107UL; // CRC poly: x^8 + x^2^x^1 + 1
  }

  if (mode == CODED) {
    NRF_RADIO->CRCCNF = (RADIO_CRCCNF_SKIPADDR_Skip << RADIO_CRCCNF_SKIPADDR_Pos) |
                        (RADIO_CRCCNF_LEN_Three << RADIO_CRCCNF_LEN_Pos);
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Ble_LR125Kbit << RADIO_MODE_MODE_Pos);
  } else if (mode == Mbit_1) {
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Nrf_1Mbit << RADIO_MODE_MODE_Pos);
  } else if (mode == Mbit_2) {
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Nrf_2Mbit << RADIO_MODE_MODE_Pos);
  }
  //channel 0 - base0 + pre0
  //channel 1 - base1 + pre1
  //channel 2 - base1 + pre2
  //channel 3 - base1 + pre3
  //channel 4 - base1 + pre4
  //channel 5 - base1 + pre5
  //channel 6 - base1 + pre6
  //channel 0 - base1 + pre7
  pre0[0] = BROADCAST_ADDR;
  pre0[1] = 0x00;
  pre0[2] = NODE;
  pre0[3] = 0x01;

  pre1[0] = 0x02;
  pre1[1] = 0x16;
  pre1[2] = 0x2a;
  pre1[3] = 0x3e;

  pre_addr0 = ((pre0[3] << 24) | (pre0[2] << 16) | (pre0[1] << 8) | (pre0[0]));
  pre_addr1 = ((pre1[3] << 24) | (pre1[2] << 16) | (pre1[1] << 8) | (pre1[0]));
  base0 = 0x11111111;
  base1 = 0x11111111;
  NRF_RADIO->BASE0 = (uint32_t)(base0);
  NRF_RADIO->BASE1 = (uint32_t)(base1);
  NRF_RADIO->PREFIX0 = (uint32_t)(pre_addr0);
  NRF_RADIO->PREFIX1 = (uint32_t)(pre_addr1);
  NRF_RADIO->RXADDRESSES = 0b00000000;
  NRF_RADIO->TXADDRESS = 0;
  NRF_RADIO->PACKETPTR = (uint32_t)&wake_mem;
  NRF_RADIO->SHORTS |= RADIO_SHORTS_ADDRESS_RSSISTART_Msk;
  NRF_RADIO->FREQUENCY = 7UL;
  NRF_RADIO->TXPOWER = powerLVL[currentPL]; // 0-10 from -40dbm to 8dbm
}
void change_next_hop_addr(uint8_t new_next_hop) {
  pre0[1] = new_next_hop;
  pre_addr0 = ((pre0[3] << 24) | (pre0[2] << 16) | (pre0[1] << 8) | (pre0[0]));
  NRF_RADIO->PREFIX0 = (uint32_t)(pre_addr0);
}
int main(void) {
  uint32_t err_code = NRF_SUCCESS;
  //Setup
  start_HFCLK();
  start_LFCLK();
  NRF_LOG_INIT(NULL);
  NRF_LOG_DEFAULT_BACKENDS_INIT();
  app_timer_init();
  NRF_LOG_INFO("######### Multi-hop sensor node start #########");
  radio_setup(CODED);
  nrf_gpio_cfg_output(ENABLE_PIN);
  i2c_init();
  i2c_scanner();
  if (i2c_device[BME680_i2c][1]) {
    bme680_calib();
    bme680_reset();
    bme680_iir_filter();
  }
  if (i2c_device[HDC2010_i2c][1]) {
    hdc2010_init();
  }
  nrf_drv_twi_disable(&m_twi);
  APP_ERROR_CHECK(err_code);
  NRF_LOG_INFO("\n\rWake interval = %d \n", WAKE_INTERVAL);
  app_timer_create(&wakeping_timer, APP_TIMER_MODE_SINGLE_SHOT, wakeup);
  app_timer_start(wakeping_timer, WAKEPING_INTERVAL, NULL);
  send_counter = SEND_INTERVAL;
  // Enter main loop.
  for (;;) {
    if (NRF_LOG_PROCESS() == false) {
      power_manage();
    }
  }
}
```

B.3 nRF52840 Gateway

Code B.9: nRF52840 Gateway Code

```
/*###########################################
  ##### Star/Multihop Gateway #############
  ###########################################*/

#define NRF_LOG_ENABLED 1
#define UART_ON
#define MULTIHOP_GATEWAY

#define START_PA_LEVEL 10
#define BROADCAST_ADDR 0xff
#include "app_error.h"
#include "app_timer.h"
#include "app_uart.h"
#include "boards.h"
#include "bsp.h"
#include "nordic_common.h"
#include "nrf_delay.h"
#include "nrf_drv_clock.h"
#include "nrf_error.h"
#include "nrf_gpio.h"
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"

// Radio Variables
uint8_t powerLVL[11] = {
    0xD8UL, //0: -40 dbm
    0xECUL, //1: -20 dbm
    0xF4UL, //2: -12 dbm
    0xFCUL, //3: -4 dbm
    0x0UL,  //4: 0 dbm
    0x2UL,  //5: 2 dbm
    0x4UL,  //6: 3 dbm
    0x5UL,  //7: 5 dbm
    0x6UL,  //8: 6 dbmS
    0x7UL,  //9: 7 dbm
    0x8UL   //10: 8 dbm
};
uint8_t currentPL = START_PA_LEVEL;
uint8_t radio_mem[20];
uint8_t data_mem[25];
uint8_t ack_mem[10];
uint8_t wake_mem[10];
uint8_t dataArray[20];
uint8_t recRSSI;
uint8_t next_hop;
enum radio_mode {
  CODED,
  Mbit_1,
  Mbit_2
};
enum data_type {
  WAKE_TYPE,
  DATA_TYPE,
  ACK_TYPE
};
uint8_t pre0[4], pre1[4];
uint32_t pre_addr0, pre_addr1, base0, base1;
bool send_wake_ping = false;
bool check_wake_timer = true;
uint32_t wake_ping_timer;

//UART variables
uint8_t prebyte = 155;
#define MAX_TEST_DATA_BYTES (15U)
#define UART_TX_BUF_SIZE 256
#define UART_RX_BUF_SIZE 256
#if defined(UART_PRESENT)
#include "nrf_uart.h"
#endif
#if defined(UARTE_PRESENT)
#include "nrf_uarte.h"
#endif
APP_TIMER_DEF(clock_timer);
uint8_t teller = 0;
void change_next_hop_addr(uint8_t new_next_hop);
void wake_ping();
void send_ack();
void start_listening();
void send_packet();
void stop_radio();
/**@brief Function for initialization oscillators.
 */
void flush_radio_mem() {
  for (int i = 0; i < sizeof(radio_mem); i++) {
    radio_mem[i] = 0;
  }
}
```

```
void flush_data_mem() {
  for (int i = 0; i < sizeof(data_mem); i++) {
    data_mem[i] = 0;
  }
}
void flush_ack_mem() {
  for (int i = 0; i < sizeof(ack_mem); i++) {
    ack_mem[i] = 0;
  }
}
uint32_t millis() {
  return (app_timer_cnt_get() / 32.768);
}
uint32_t micros() {
  return (app_timer_cnt_get() / 0.032786);
}
void uart_error_handle(app_uart_evt_t *p_event) {
  if (p_event->evt_type == APP_UART_COMMUNICATION_ERROR) {
    APP_ERROR_HANDLER(p_event->data.error_communication);
  } else if (p_event->evt_type == APP_UART_FIFO_ERROR) {
    APP_ERROR_HANDLER(p_event->data.error_code);
  }
}
void start_LFCLK() {
  /* Start low frequency crystal oscillator for app_timer(used by bsp)*/
  NRF_CLOCK->LFCLKSRC = (CLOCK_LFCLKSRC_SRC_Xtal << CLOCK_LFCLKSRC_SRC_Pos);
  NRF_CLOCK->EVENTS_LFCLKSTARTED = 0;
  NRF_CLOCK->TASKS_LFCLKSTART = 1;

  while (NRF_CLOCK->EVENTS_LFCLKSTARTED == 0) {
    // Do nothing.
  }
}
void start_HFCLK(void) {
  NRF_CLOCK->EVENTS_HFCLKSTARTED = 0;
  NRF_CLOCK->TASKS_HFCLKSTART = 1;
  /* Wait for the external oscillator to start up */
  while (NRF_CLOCK->EVENTS_HFCLKSTARTED == 0) {
    // Do nothing.
  }
}

/**@brief Function for reading packet.
 */
void start_listening() {

  //Start Listening
  NRF_RADIO->EVENTS_READY = 0U;
  // Enable radio and wait for ready
  NRF_RADIO->TASKS_RXEN = 1U;

  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->EVENTS_END = 0U;
  // Start listening and wait for address received event
  NRF_RADIO->TASKS_START = 1U;
}
void wake_ping() {
  stop_radio();
  NRF_RADIO->TXPOWER = powerLVL[START_PA_LEVEL - 1];
  NRF_RADIO->TXADDRESS = 0;
  NRF_RADIO->PACKETPTR = (uint32_t)&wake_mem;
  NRF_LOG_INFO("wake ping %d", millis());
  uint8_t byte = 2;
  wake_mem[byte++] = WAKE_TYPE; // MY_ID (sink)
  wake_mem[byte++] = 0;         // MY_ID (sink)
  wake_mem[byte++] = 0;         // my_hops_to_sink
  wake_mem[0] = byte - 2;
  wake_mem[1] = 0;
  send_packet();
  wake_ping_timer = millis() + 1000;
  if (wake_ping_timer > 511995) {
    wake_ping_timer = 0;
    check_wake_timer = false;
  }
  stop_radio();
  NRF_RADIO->TXPOWER = powerLVL[START_PA_LEVEL];
  NRF_RADIO->PACKETPTR = (uint32_t)&data_mem;
  NRF_RADIO->RXADDRESSES = 0b00000100;
  NRF_RADIO->EVENTS_END = 0;
  start_listening();
}
void wait_for_data() {
  NRF_RADIO->PACKETPTR = (uint32_t)&data_mem;
  NRF_RADIO->RXADDRESSES = 0b00000100;
  start_listening();
  while (NRF_RADIO->EVENTS_END == 0U) {
  // wait
#ifdef MULTIHOP_GATEWAY
    if (wake_ping_timer < millis()) {
      wake_ping();
      NRF_LOG_FLUSH();
    }
#endif
  }
  NRF_RADIO->EVENTS_END = 0;
  stop_radio();
  if ((NRF_RADIO->CRCSTATUS == 1U) && (NRF_RADIO->RXMATCH == 2)) {
    if ((data_mem[2] == DATA_TYPE) && (data_mem[3] == 0)) {
      send_ack();
    }
  }
  for (int i = 0; i < data_mem[0] + 2; i++) {
    NRF_LOG_INFO("%d", data_mem[i]);
  }
#ifdef UART_ON
  app_uart_put(155);
  for (int i = 0; i < data_mem[0] + 2; i++) {
    app_uart_put(data_mem[i]);
  }
#endif
  flush_data_mem();
}
```

```
void send_ack() {
#ifdef MULTIHOP_GATEWAY
  change_next_hop_addr(data_mem[4]);
  NRF_RADIO->PACKETPTR = (uint32_t)&ack_mem;
  NRF_RADIO->TXADDRESS = 1;
#endif
  uint8_t byte = 2;
  ack_mem[byte++] = ACK_TYPE;
  ack_mem[byte++] = data_mem[4]; // ID
#ifndef MULTIHOP_GATEWAY
  radio_mem[byte++] = recRSSI;
#endif
  ack_mem[0] = byte - 2;
  ack_mem[1] = 0;
  nrf_delay_us(10);
  send_packet();
  flush_ack_mem();
  stop_radio();
}
void stop_radio() {
  NRF_RADIO->EVENTS_DISABLED = 0U;
  // Disable radio
  NRF_RADIO->TASKS_DISABLE = 1U;

  while (NRF_RADIO->EVENTS_DISABLED == 0U) {
    // wait
  }
}
void send_packet() {
  // send the packet:
  NRF_RADIO->TASKS_RXEN = 0;
  NRF_RADIO->EVENTS_READY = 0U;
  NRF_RADIO->TASKS_TXEN = 1;

  while (NRF_RADIO->EVENTS_READY == 0U) {
    // wait
  }
  NRF_RADIO->EVENTS_END = 0U;
  NRF_RADIO->TASKS_START = 1U;

  while (NRF_RADIO->EVENTS_END == 0U) {
    // wait
  }
}

void radio_setup(uint8_t mode) {

  if (mode == CODED) {
    NRF_LOG_INFO("Using Coded!");
    uint32_t preamble_mask = (RADIO_PCNF0_PLEN_LongRange << RADIO_PCNF0_PLEN_Pos) |
                             (2 << RADIO_PCNF0_CILEN_Pos) |
                             (3 << RADIO_PCNF0_TERMLEN_Pos);

    NRF_RADIO->PCNF0 = (1UL << RADIO_PCNF0_S1LEN_Pos) |
                       (2UL << RADIO_PCNF0_S0LEN_Pos) |
                       (8UL << RADIO_PCNF0_LFLEN_Pos) |
                       preamble_mask;

    NRF_RADIO->PCNF1 = (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
                       (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos) |
                       (3UL << RADIO_PCNF1_BALEN_Pos) |
                       (0 << RADIO_PCNF1_STATLEN_Pos) |
                       (250 << RADIO_PCNF1_MAXLEN_Pos);
  } else {
    NRF_RADIO->PCNF0 = (0 << RADIO_PCNF0_S1LEN_Pos) |
                       (0 << RADIO_PCNF0_S0LEN_Pos) |
                       (8 << RADIO_PCNF0_LFLEN_Pos);

    NRF_RADIO->PCNF1 = (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
                       (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos) |
                       (4 << RADIO_PCNF1_BALEN_Pos) |
                       (0 << RADIO_PCNF1_STATLEN_Pos) |
                       (200 << RADIO_PCNF1_MAXLEN_Pos);
  }
  NRF_RADIO->CRCCNF = (RADIO_CRCCNF_LEN_Two << RADIO_CRCCNF_LEN_Pos); // checksum bits
  if ((NRF_RADIO->CRCCNF & RADIO_CRCCNF_LEN_Msk) ==
      (RADIO_CRCCNF_LEN_Two << RADIO_CRCCNF_LEN_Pos)) {
    NRF_RADIO->CRCINIT = 0xFFFFUL;  // Initial value
    NRF_RADIO->CRCPOLY = 0x11021UL; // CRC poly: x^16 + x^12^x^5 + 1
  } else if ((NRF_RADIO->CRCCNF & RADIO_CRCCNF_LEN_Msk) ==
             (RADIO_CRCCNF_LEN_One << RADIO_CRCCNF_LEN_Pos)) {
    NRF_RADIO->CRCINIT = 0xFFUL;  // Initial value
    NRF_RADIO->CRCPOLY = 0x107UL; // CRC poly: x^8 + x^2^x^1 + 1
  }

  if (mode == CODED) {
    NRF_RADIO->CRCCNF = (RADIO_CRCCNF_SKIPADDR_Skip << RADIO_CRCCNF_SKIPADDR_Pos) |
                        (RADIO_CRCCNF_LEN_Three << RADIO_CRCCNF_LEN_Pos);
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Ble_LR125Kbit << RADIO_MODE_MODE_Pos);
  } else if (mode == Mbit_1) {
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Nrf_1Mbit << RADIO_MODE_MODE_Pos);
  } else if (mode == Mbit_2) {
    NRF_RADIO->MODE = (RADIO_MODE_MODE_Nrf_2Mbit << RADIO_MODE_MODE_Pos);
  }

#ifdef MULTIHOP_GATEWAY
  pre0[0] = BROADCAST_ADDR;
  pre0[1] = next_hop;
  pre0[2] = 0x00;
  pre0[3] = 0x01;
#else
  pre0[0] = 0x02;
  pre0[1] = 0x11;
  pre0[2] = 0x22;
  pre0[3] = 0x01;
#endif
  pre1[0] = 0x02;
  pre1[1] = 0x16;
  pre1[2] = 0x2a;
  pre1[3] = 0x3e;

  pre_addr0 = ((pre0[3] << 24) | (pre0[2] << 16) | (pre0[1] << 8) | (pre0[0]));
  pre_addr1 = ((pre1[3] << 24) | (pre1[2] << 16) | (pre1[1] << 8) | (pre1[0]));
  base0 = 0x11111111;
```

```c
  base1 = 0x11111111;

  NRF_RADIO->BASE0 = (uint32_t)(base0);
  NRF_RADIO->BASE1 = (uint32_t)(base1);
  NRF_RADIO->PREFIX0 = (uint32_t)(pre_addr0);
  NRF_RADIO->PREFIX1 = (uint32_t)(pre_addr1);
  NRF_RADIO->TXADDRESS = 0;
  NRF_RADIO->RXADDRESSES = 0b00000000;
  NRF_RADIO->PACKETPTR = (uint32_t)&radio_mem;
  NRF_RADIO->SHORTS |= RADIO_SHORTS_ADDRESS_RSSISTART_Msk;
  NRF_RADIO->FREQUENCY = 7UL;
  NRF_RADIO->TXPOWER = powerLVL[currentPL]; // 0-10 from -40dbm to 8dbm
}
void change_next_hop_addr(uint8_t new_next_hop) {
  pre0[1] = new_next_hop;
  pre_addr0 = ((pre0[3] << 24) | (pre0[2] << 16) | (pre0[1] << 8) | (pre0[0]));
  NRF_RADIO->PREFIX0 = (uint32_t)(pre_addr0);
}
int main(void) {
  uint32_t err_code = NRF_SUCCESS;
  start_HFCLK();
  start_LFCLK();
  NRF_LOG_INIT(NULL);
  NRF_LOG_DEFAULT_BACKENDS_INIT();
  app_timer_init();
  NRF_RTC1->TASKS_START = 1;

  NRF_LOG_INFO("\n\r################Gateway Start!################\n");
#ifdef UART_ON
  const app_uart_comm_params_t comm_params =
      {
          RX_PIN_NUMBER,
          TX_PIN_NUMBER,
          CTS_PIN_NUMBER,
          RTS_PIN_NUMBER,
          APP_UART_FLOW_CONTROL_DISABLED,
          false,
          NRF_UART_BAUDRATE_115200};

  APP_UART_FIFO_INIT(&comm_params,
      UART_RX_BUF_SIZE,
      UART_TX_BUF_SIZE,
      uart_error_handle,
      APP_IRQ_PRIORITY_LOWEST,
      err_code);
#endif
  radio_setup(CODED);
  while (1) {
    wait_for_data();
    NRF_LOG_FLUSH();
  }
}
```

B.4 BME680

B.4.1 BME680.h

Code B.10: BME680.h Code

```
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
#include "nrf_drv_twi.h"
#define BME_ADDR  (0x77)
#define TWI_INSTANCE_ID 0
#define GAS_HEAT_DURATION 200 // 200 //  in millisec. from 63-4032
#define GAS_HEAT_TEMP 300 //200 // from 200-400c
static const nrf_drv_twi_t m_twi = NRF_DRV_TWI_INSTANCE(TWI_INSTANCE_ID);
typedef union {
        uint8_t rawData;
        struct {
                uint8_t nb_conv  : 4;
                uint8_t run_gas  : 1;
                uint8_t reserved : 3;
        };
} ClosedCube_BME680_Heater_Profile;
struct bme680_cal_temp {
        uint16_t t1, t2;
        uint8_t t3;
};
struct bme680_cal_pres {
        uint16_t p1;
        int16_t p2, p4, p5, p8, p9;
        int8_t p3, p6, p7, p10;
};
struct bme680_cal_hum {
        uint16_t h1, h2;
        int8_t h3, h4, h5, h7;
        uint8_t h6;
};
struct bme680_cal_dev {
        int32_t tfine;
        uint8_t amb_temp;
        uint8_t res_heat_range;
        int8_t res_heat_val;
        int8_t range_sw_err;
};
struct bme680_cal_gas {
        int8_t gh1, gh3;
        int16_t gh2;
};
static struct bme680_cal_temp _calib_temp;
static struct bme680_cal_pres _calib_pres;
static struct bme680_cal_hum _calib_hum;
static struct bme680_cal_dev _calib_dev;
static struct bme680_cal_gas _calib_gas;

static uint8_t bme_data[8];
static uint32_t bmeGas;
static uint8_t pres_msb,pres_lsb,pres_xlsb,temp_msb,temp_lsb,temp_xlsb,hum_msb,hum_lsb;
static uint8_t chipID;
static double tempD,humD,presD;
static uint16_t bmeHum16,bmePres16,bmeGas16;
static int16_t bmeTemp16;

uint8_t bme680_read_reg(uint8_t reg);
void bme680_set_wake_period(uint8_t num);
uint8_t bme680_calc_iaq(void);
uint8_t bme680_gas_temp_calc(uint16_t heaterTemperature);
uint8_t bme680_gas_dur_calc(uint16_t heaterDuration);
void bme680_gas_on(void);
void bme680_gas_off(void);
uint32_t bme680_gas_read(void);
void bme680_ask_data(void);
void bme680_iir_filter(void);
void bme680_reset(void);
uint8_t bme680_chip_id(void);
void bme680_calib(void);
void bme680_get_hum(void);
void bme680_get_pres(void);
double bme680_temp_calc(void);
double bme680_hum_calc(void);
double bme680_pres_calc(void);
void bme680_get_data(void);
```

B.4.2 BME680.c Code

Code B.11: BME680.c Code

```
#include <nrf_bme680.h>

ClosedCube_BME680_Heater_Profile profile;

const uint32_t lookupTable1[16] = {2147483647, 2147483647, 2147483647, 2147483647, 2147483647,
    2126008810, 2147483647, 2130303777, 2147483647, 2147483647,
    2143188679, 2136746228, 2147483647, 2126008810, 2147483647,
    2147483647};

const uint32_t lookupTable2[16] = {4096000000, 2048000000, 1024000000, 512000000, 255744255,
    127110228, 64000000, 32258064, 16016016, 8000000,
    4000000, 2000000, 1000000, 500000, 250000,
    125000};

uint8_t bme680_read_reg(uint8_t reg) {
  uint8_t reg_data;
  nrf_drv_twi_tx(&m_twi, BME_ADDR, &reg, 1, false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &reg_data, sizeof(reg_data));
  //NRF_LOG_INFO("reg %d contains %d",reg,reg_data);
  return reg_data;
}
void bme680_set_wake_period(uint8_t num) {
  uint8_t wake_data[2] = {0x71, num};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, wake_data, 2, false);
}
uint8_t bme680_calc_iaq(void) {
  uint8_t iaq;
  float hum_score, gas_score;
  float gas_reference = 250000;
  float hum_reference = 40;
  //Calculate humidity contribution to IAQ index
  float current_humidity = humD;
  if (current_humidity >= 38 && current_humidity <= 42) {
    hum_score = 0.25 * 100; // Humidity +/-5% around optimum
  } else {                  //sub-optimal
    if (current_humidity < 38) {
      hum_score = 0.25 / hum_reference * current_humidity * 100;
    }

    else {
      hum_score = ((-0.25 / (100 - hum_reference) * current_humidity) + 0.416666) * 100;
    }
  }
  //Calculate gas contribution to IAQ index
  gas_reference = bmeGas;
  int gas_lower_limit = 50000;  // Bad air quality limit
  int gas_upper_limit = 500000; // Good air quality limit
  if (gas_reference > gas_upper_limit) {
    gas_reference = gas_upper_limit;
  }
  if (gas_reference < gas_lower_limit) {
    gas_reference = gas_lower_limit;
  }
  gas_score = (0.75 / (gas_upper_limit - gas_lower_limit) * gas_reference - (gas_lower_limit * (0.75 / (gas_upper_limit

  //Combine results for the final IAQ index value (0-100% where 100% is good quality air)
  float score = hum_score + gas_score;
  score = (100 - score) * 5;
  //NRF_LOG_INFO("score = %d",score);
  if (score >= 301) {
    iaq = 5;
  } //{NRF_LOG_INFO("Hazardous");}
  else if (score >= 201 && score <= 300) {
    iaq = 4;
  } // {NRF_LOG_INFO("Very Unhealthy");}
  else if (score >= 176 && score <= 200) {
    iaq = 3;
  } // {NRF_LOG_INFO("Unhealthy");}
  else if (score >= 151 && score <= 175) {
    iaq = 2;
  } // {NRF_LOG_INFO("Unhealthy for Sensitive Groups");}
  else if (score >= 51 && score <= 150) {
    iaq = 1;
  } // {NRF_LOG_INFO("Moderate");}
  else if (score >= 00 && score <= 50) {
    iaq = 0;
  } //{NRF_LOG_INFO("Good");}
  return iaq;
}
uint8_t bme680_gas_temp_calc(uint16_t heaterTemperature) {
  int32_t var1, var2, var3, var4, var5;
  int32_t heatr_res_x100;

  if (heaterTemperature < 200)
    heaterTemperature = 200;
  else if (heaterTemperature > 400)
    heaterTemperature = 400;

  var1 = (((int32_t)_calib_dev.amb_temp * _calib_gas.gh3) / 1000) * 256;
  var2 = (_calib_gas.gh1 + 784) * (((((_calib_gas.gh2 + 154009) * heaterTemperature * 5) / 100) + 3276800) / 10);
  var3 = var1 + (var2 / 2);
  var4 = (var3 / (_calib_dev.res_heat_range + 4));
  var5 = (131 * _calib_dev.res_heat_val) + 65536;
  heatr_res_x100 = (int32_t)(((var4 / var5) - 250) * 34);
  return (uint8_t)((heatr_res_x100 + 50) / 100);
}
uint8_t bme680_gas_dur_calc(uint16_t heaterDuration) {
  uint8_t factor = 0;
  uint8_t durval;

  if (heaterDuration >= 0xfc0) {
    durval = 0xff;
  } else {
    while (heaterDuration > 0x3F) {
      heaterDuration = heaterDuration / 4;
      factor += 1;
    }
    durval = (uint8_t)(heaterDuration + (factor * 64));
  }

  return durval;
}
void bme680_gas_on(void) {
  uint8_t gas_on_data1[4] = {0x5a, bme680_gas_temp_calc(GAS_HEAT_TEMP), 0x64, bme680_gas_dur_calc(GAS_HEAT_DURATION)};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, gas_on_data1, 4, false);
  profile.nb_conv = 0;
  profile.run_gas = 1;

  uint8_t gas_on_data2[2] = {0x71, profile.rawData};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, gas_on_data2, 2, false);
}
void bme680_gas_off(void) {
  profile.nb_conv = 0;
  profile.run_gas = 0;
  uint8_t gas_off_data[2] = {0x71, profile.rawData};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, gas_off_data, 2, false);
}

uint32_t bme680_gas_read(void) {
  typedef union {
    uint8_t raw;
    union {
      uint8_t range : 2;
      uint8_t lsb : 6;
    };
  } gas_lsb_t;

  gas_lsb_t gas_lsb;

  uint8_t gas_msb = bme680_read_reg(0x2A);
  gas_lsb.raw = bme680_read_reg(0x2B);

  uint16_t gas_raw = gas_msb << 8 | gas_lsb.lsb;

  int64_t var1, var2, var3;

  var1 = (int64_t)((1340 + (5 * (int64_t)_calib_dev.range_sw_err)) * ((int64_t)lookupTable1[gas_lsb.range])) / 65536;
  var2 = (((int64_t)((int64_t)gas_raw * 32768) - (int64_t)(16777216)) + var1);
  var3 = (((int64_t)lookupTable2[gas_lsb.range] * (int64_t)var1) / 512);

  return (uint32_t)((var3 + ((int64_t)var2 / 2)) / (int64_t)var2);
}
void bme680_ask_data(void) {

  uint8_t forced_reg[2] = {0x72, 0x74};
  uint8_t forced_data;
  //forced_data = 0x02 & 0b00000111; //hum oversampling
  forced_data = 0x01 & 0b00000111; //hum oversampling
  forced_reg[1] = forced_data;
  nrf_drv_twi_tx(&m_twi, BME_ADDR, forced_reg, 2, false);
  //forced_data = (0x01 << 5) & 0b11100000;  // temp oversampling
  //forced_data |= (0x02 << 2) & 0b00011100; // pres oversampling
  forced_data = (0x01 << 5) & 0b11100000;  // temp oversampling
  forced_data |= (0x01 << 2) & 0b00011100; // pres oversampling
  forced_data |= 0x01 & 0b00000011;        // operation mode
  forced_reg[0] = 0x74;
  forced_reg[1] = forced_data;
  nrf_drv_twi_tx(&m_twi, BME_ADDR, forced_reg, 2, false);
}
void bme680_iir_filter(void) {
  uint8_t iir_data[2] = {0x75, 0x00};
  //iir_data[1] = (0b100 << 2) & 0b00011100;
  iir_data[1] = (0b001 << 2) & 0b00011100;
  nrf_drv_twi_tx(&m_twi, BME_ADDR, iir_data, sizeof(iir_data), false);
}
void bme680_reset(void) {
  uint8_t reset_reg = 0xE0;
  uint8_t reset_out;

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &reset_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &reset_out, sizeof(reset_out));
}
uint8_t bme680_chip_id(void) {
  uint8_t chipID_reg = 0xD0;

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &chipID_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &chipID, sizeof(chipID));
  return chipID;
}
void bme680_calib(void) {
  uint8_t cal1[25];
  uint8_t cal2[16];
  uint8_t cal1_reg = 0x89;
  uint8_t cal2_reg = 0xE1;

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &cal1_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, cal1, sizeof(cal1));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &cal2_reg, 1, true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, cal2, sizeof(cal2));

  _calib_temp.t1 = cal2[9] << 8 | cal2[8];
  _calib_temp.t2 = cal1[2] << 8 | cal1[1];
  _calib_temp.t3 = cal1[3];
  _calib_hum.h1 = cal2[2] << 4 | (cal2[1] & 0x0F);
  _calib_hum.h2 = cal2[0] << 4 | cal2[1];
  _calib_hum.h3 = cal2[3];
  _calib_hum.h4 = cal2[4];
  _calib_hum.h5 = cal2[5];
  _calib_hum.h6 = cal2[6];
  _calib_hum.h7 = cal2[7];

  _calib_pres.p1 = cal1[6] << 8 | cal1[5];
  _calib_pres.p2 = cal1[8] << 8 | cal1[7];
  _calib_pres.p3 = cal1[9];
  _calib_pres.p4 = cal1[12] << 8 | cal1[11];
  _calib_pres.p5 = cal1[14] << 8 | cal1[13];
  _calib_pres.p6 = cal1[16];
  _calib_pres.p7 = cal1[15];
  _calib_pres.p8 = cal1[20] << 8 | cal1[19];
  _calib_pres.p9 = cal1[22] << 8 | cal1[21];
  _calib_pres.p10 = cal1[23];

  _calib_gas.gh1 = cal2[14];
  _calib_gas.gh2 = cal2[12] << 8 | cal2[13];
  _calib_gas.gh3 = cal2[15];

  uint8_t heat_range_reg = 0x02;
  uint8_t res_heat_val_reg = 0x00;
  uint8_t sw_err_reg = 0x04;

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &heat_range_reg, sizeof(heat_range_reg), true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &_calib_dev.res_heat_range, sizeof(_calib_dev.res_heat_range));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &res_heat_val_reg, sizeof(res_heat_val_reg), true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &_calib_dev.res_heat_val, sizeof(_calib_dev.res_heat_val));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &sw_err_reg, sizeof(sw_err_reg), true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &_calib_dev.range_sw_err, sizeof(_calib_dev.range_sw_err));
}
void bme680_get_temp(void) {
  uint8_t temp_reg[3] = {0x22, 0x23, 0x24};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, &temp_reg[0], sizeof(temp_reg[0]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &temp_msb, sizeof(temp_msb));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &temp_reg[1], sizeof(temp_reg[1]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &temp_lsb, sizeof(temp_lsb));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &temp_reg[2], sizeof(temp_reg[2]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &temp_xlsb, sizeof(temp_xlsb));
}
```

```
void bme680_get_hum(void) {
  uint8_t hum_reg[2] = {0x25, 0x26};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, &hum_reg[0], sizeof(hum_reg[0]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &hum_msb, sizeof(hum_msb));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &hum_reg[1], sizeof(hum_reg[1]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &hum_lsb, sizeof(hum_lsb));
}

void bme680_get_pres(void) {
  uint8_t pres_reg[3] = {0x1f, 0x20, 0x21};
  nrf_drv_twi_tx(&m_twi, BME_ADDR, &pres_reg[0], sizeof(pres_reg[0]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &pres_msb, sizeof(pres_msb));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &pres_reg[1], sizeof(pres_reg[1]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &pres_lsb, sizeof(pres_lsb));

  nrf_drv_twi_tx(&m_twi, BME_ADDR, &pres_reg[2], sizeof(pres_reg[2]), false);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, &pres_xlsb, sizeof(pres_xlsb));
}

double bme680_temp_calc(void) {
  uint32_t temp_raw = ((uint32_t)temp_msb << 12) | ((uint32_t)temp_lsb << 4) | ((uint32_t)temp_xlsb >> 4);

  uint32_t var1, var2, var3, calc_temp;

  var1 = ((int32_t)temp_raw / 8) - ((int32_t)_calib_temp.t1 * 2);
  var2 = (var1 * (int32_t)_calib_temp.t2) / 2048;
  var3 = ((var1 / 2) * (var1 / 2)) / 4096;
  var3 = ((var3) * ((int32_t)_calib_temp.t3 * 16)) / 16384;
  _calib_dev.tfine = (int32_t)(var2 + var3);
  calc_temp = (int16_t)(((_calib_dev.tfine * 5) + 128) / 256);
  tempD = calc_temp / 100.0;
  return tempD;
}
double bme680_hum_calc(void) {
  uint16_t hum_raw = hum_msb << 8 | hum_lsb;

  int32_t var1, var2, var3, var4, var5, var6, temp, calc_hum;

  temp = (((int32_t)_calib_dev.tfine * 5) + 128) / 256;
  var1 = (int32_t)(hum_raw - ((int32_t)((int32_t)_calib_hum.h1 * 16))) - (((temp * (int32_t)_calib_hum.h3) / ((int32_t)
  var2 = ((int32_t)_calib_hum.h2 * (((temp * (int32_t)_calib_hum.h4) / ((int32_t)100)) + (((temp * ((temp * (int32_t)_c
  var3 = var1 * var2;
  var4 = (int32_t)_calib_hum.h6 * 128;
  var4 = ((var4) + ((temp * (int32_t)_calib_hum.h7) / ((int32_t)100))) / 16;
  var5 = ((var3 / 16384) * (var3 / 16384)) / 1024;
  var6 = (var4 * var5) / 2;
  calc_hum = (((var3 + var6) / 1024) * ((int32_t)1000)) / 4096 / 1000.0;

  if (calc_hum > 100)
    calc_hum = 100;
  else if (calc_hum < 0)
    calc_hum = 0;

  humD = calc_hum;
  return humD;
}
double bme680_pres_calc(void) {
  uint32_t pres_raw = ((uint32_t)pres_msb << 12) | ((uint32_t)pres_lsb << 4) | ((uint32_t)pres_xlsb >> 4);

  int32_t var1, var2, var3, calc_pres;

  var1 = (((int32_t)_calib_dev.tfine) / 2) - 64000;
  var2 = ((var1 / 4) * (var1 / 4)) / 2048;
  var2 = ((var2) * (int32_t)_calib_pres.p6) / 4;
  var2 = var2 + ((var1 * (int32_t)_calib_pres.p5) * 2);
  var2 = (var2 / 4) + ((int32_t)_calib_pres.p4 * 65536);
  var1 = ((var1 / 4) * (var1 / 4)) / 8192;
  var1 = (((var1) * ((int32_t)_calib_pres.p3 * 32)) / 8) + (((int32_t)_calib_pres.p2 * var1) / 2);
  var1 = var1 / 262144;
  var1 = ((32768 + var1) * (int32_t)_calib_pres.p1) / 32768;
  calc_pres = (int32_t)(1048576 - pres_raw);
  calc_pres = (int32_t)((calc_pres - (var2 / 4096)) * (3125));
  calc_pres = ((calc_pres / var1) * 2);
  var1 = ((int32_t)_calib_pres.p9 * (int32_t)(((calc_pres / 8) * (calc_pres / 8)) / 8192)) / 4096;
  var2 = ((int32_t)(calc_pres / 4) * (int32_t)_calib_pres.p8) / 8192;
  var3 = ((int32_t)(calc_pres / 256) * (int32_t)(calc_pres / 256) * (int32_t)(calc_pres / 256) * (int32_t)_calib_pres.p
  calc_pres = (int32_t)(calc_pres) + ((var1 + var2 + var3 + ((int32_t)_calib_pres.p7 * 128)) / 16);

  presD = calc_pres / 100.0;
  return presD;
}
void bme680_get_data(void) {
  uint8_t data_reg = 0x1F; // 0x1f
  uint64_t data64;
  for (int i = 0; i < sizeof(bme_data); i++) {
    bme_data[i] = 0x00;
  }
  nrf_drv_twi_tx(&m_twi, BME_ADDR, &data_reg, sizeof(data_reg), true);
  nrf_drv_twi_rx(&m_twi, BME_ADDR, bme_data, sizeof(bme_data));

  pres_msb = bme_data[0];
  pres_lsb = bme_data[1];
  pres_xlsb = bme_data[2];
  temp_msb = bme_data[3];
  temp_lsb = bme_data[4];
  temp_xlsb = bme_data[5];
  hum_msb = bme_data[6];
  hum_lsb = bme_data[7];
}
```

Appendix C

Soldering Method

As most of the ICs used for the project are SMD type and have pads on the bottom side, soldering by hand is impossible. Therefore, soldering is done by applying solder paste to the PCBs, placing all the components, and then heating everything up to the necessary temperatures in a reflow oven.

UiA has a CNC machine for PCBs, however not with the possibility for soldermask and silkscreen, resulting in a higher chance for short circuits between pads, and therefore higher difficulty in soldering. Therefore, PCBs are ordered from a Chinese supplier who provides sufficient quality for the sensor nodes at very low costs.

In order to simplify the application of solder paste, a stencil is created by using the laser functionality of the Snapmaker. Glossy photo paper is used in order to reduce the cost compared to metal sheet stencils available from external suppliers. The process of laser cutting stencils is depicted in figure C.1

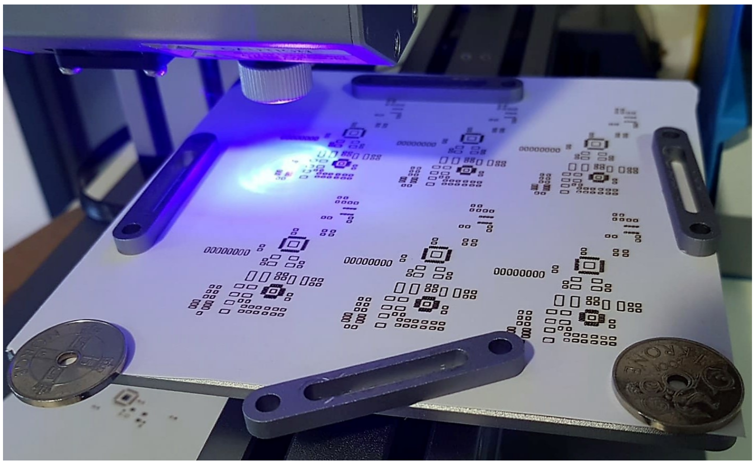


Figure C.1: Cutting stencils with the Snapmaker

By utilizing the glossy side of the paper sheets, a spatula can be used to easily apply the solder paste with little friction. A leaded solder paste is used to allow for lower temperatures in the reflow process. Figure C.2 shows the solder paste application setup for the project. A jig is made by mounting old PCBs with perfectly straight edges to a table by using tape in order to hold the PCB to be soldered. The stencil is then lined up with the pads of the PCB, and a spatula is used to evenly apply the solder paste.

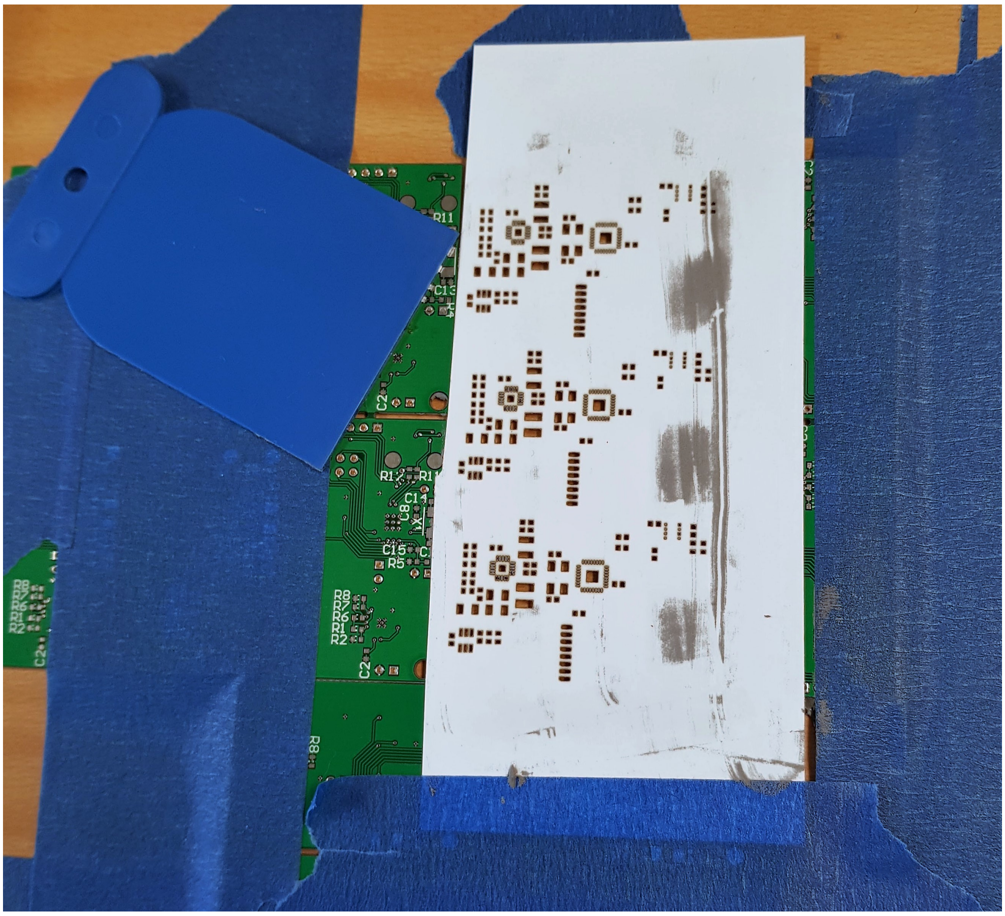


Figure C.2: Applying solder paste to the PCB

A manual pick and place machine available at UiA is used for placing the components in a precise manner on the PCB. Figure C.3 shows the machine during the placing process. The machine has a vacuum nozzle to pick up and hold the components, which is automatically released when the components are in contact with the PCB with sufficient pressure. The arm holding the nozzle can be moved freely in the X and Y and Z axes.

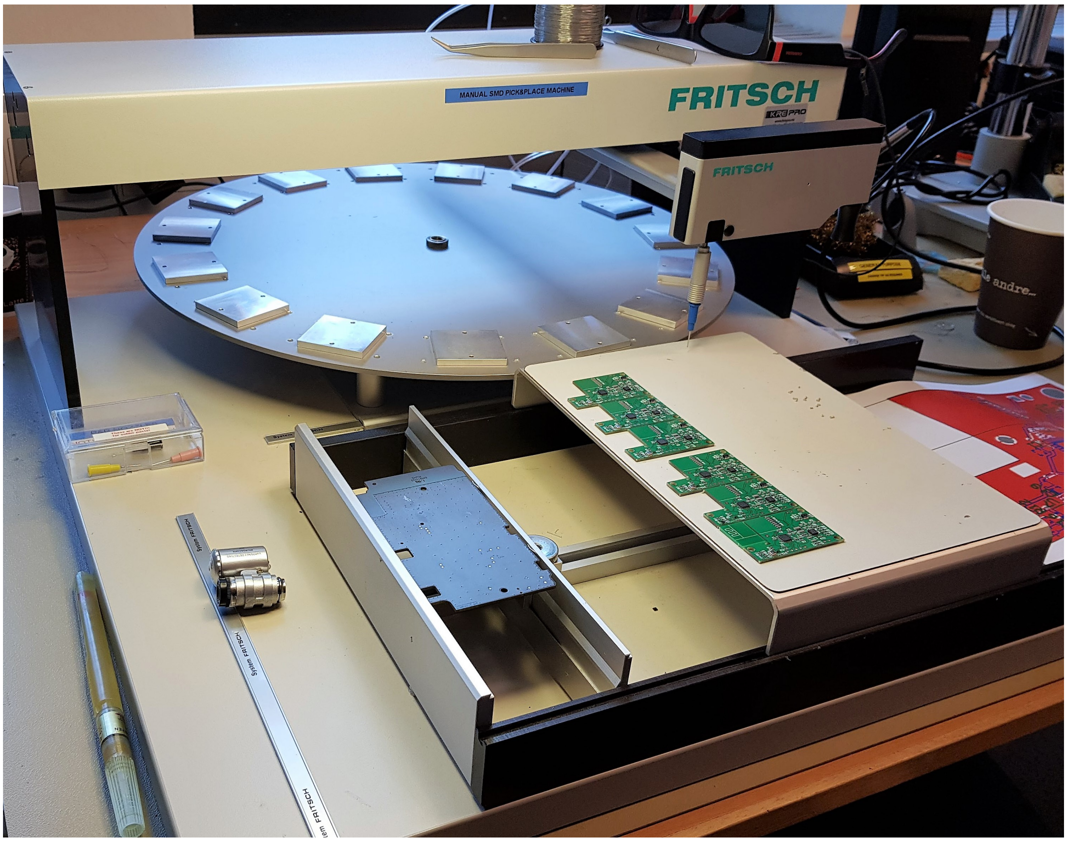


Figure C.3: Manual pick and place machine

Figure C.4 shows the PCBs with the solder paste applied during the pick and place process, with some ICs already placed.

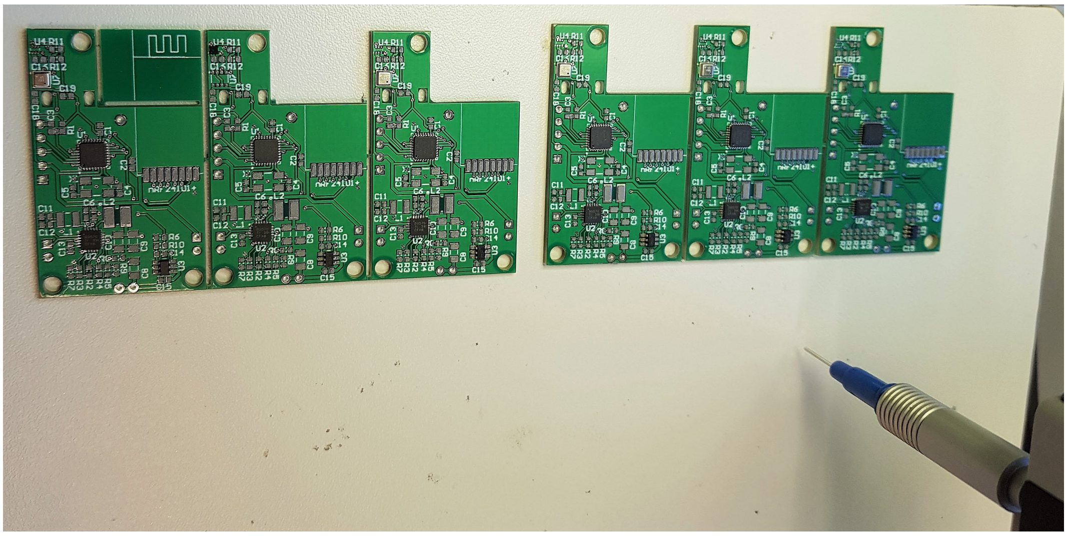


Figure C.4: PCB with solder paste during the pick and place process

A reflow oven available at UiA is used for soldering the PCBs. A pre-heat temperature of 140°C for 150s is used. In order to ensure that all the solder paste is properly and evenly melted during the reflow process, a temperature of 260°C, which is the recommended temperature for the ICs used in the sensor nodes, is used for 170s before being cooled down to room temperature.

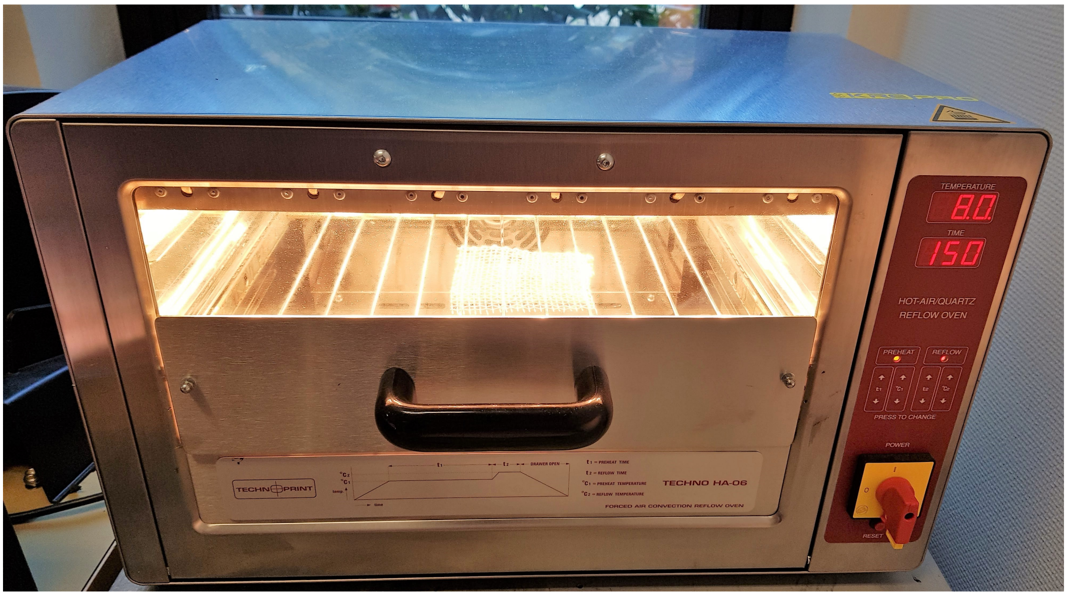


Figure C.5: Reflow oven

Figure C.6 shows the sensor node PCBs with most SMD components soldered in place by reflow soldering. The through-hole components, as well as some of the SMD components, are hand soldered after the reflow process.

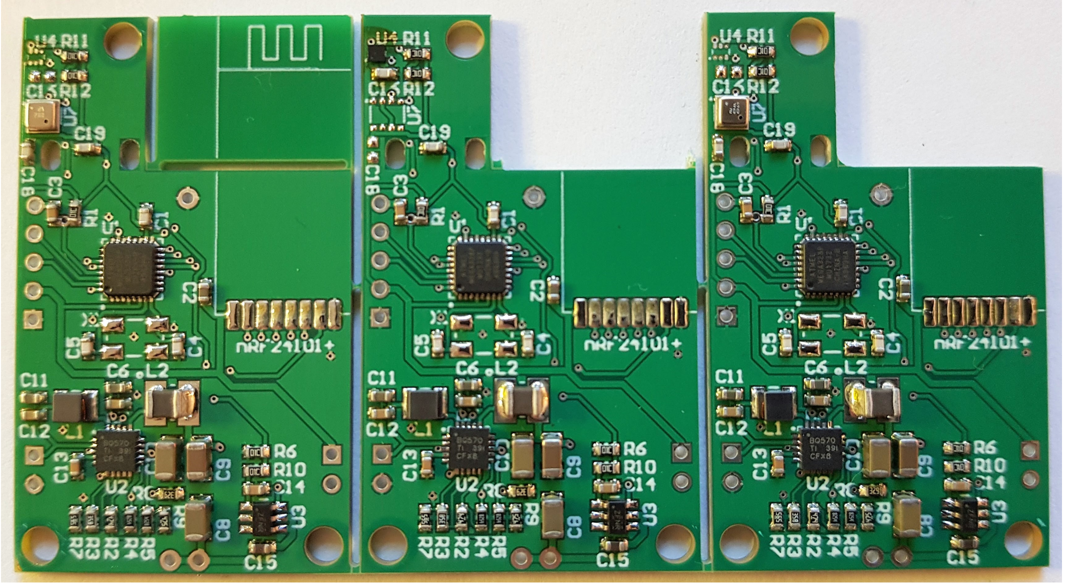
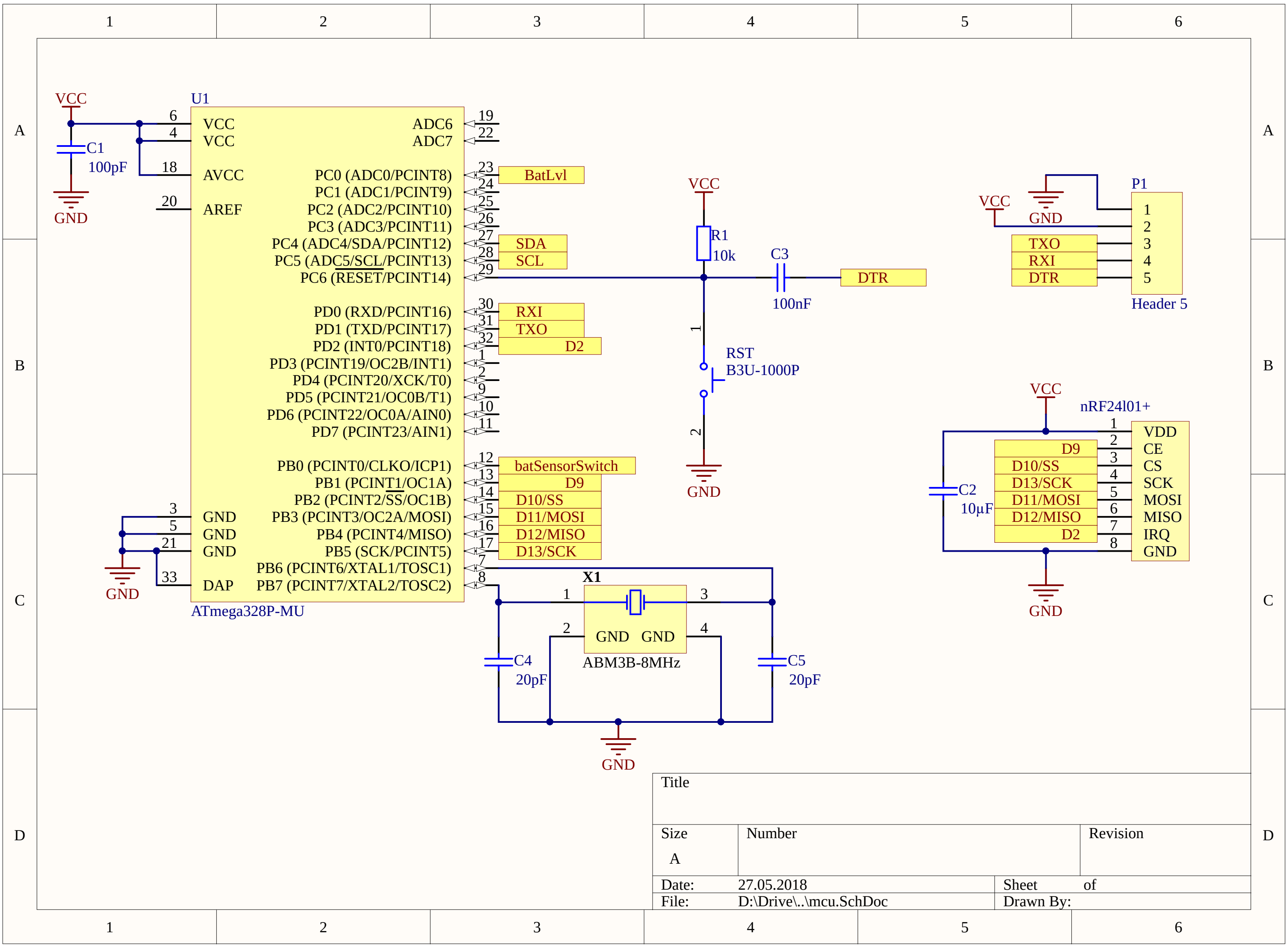


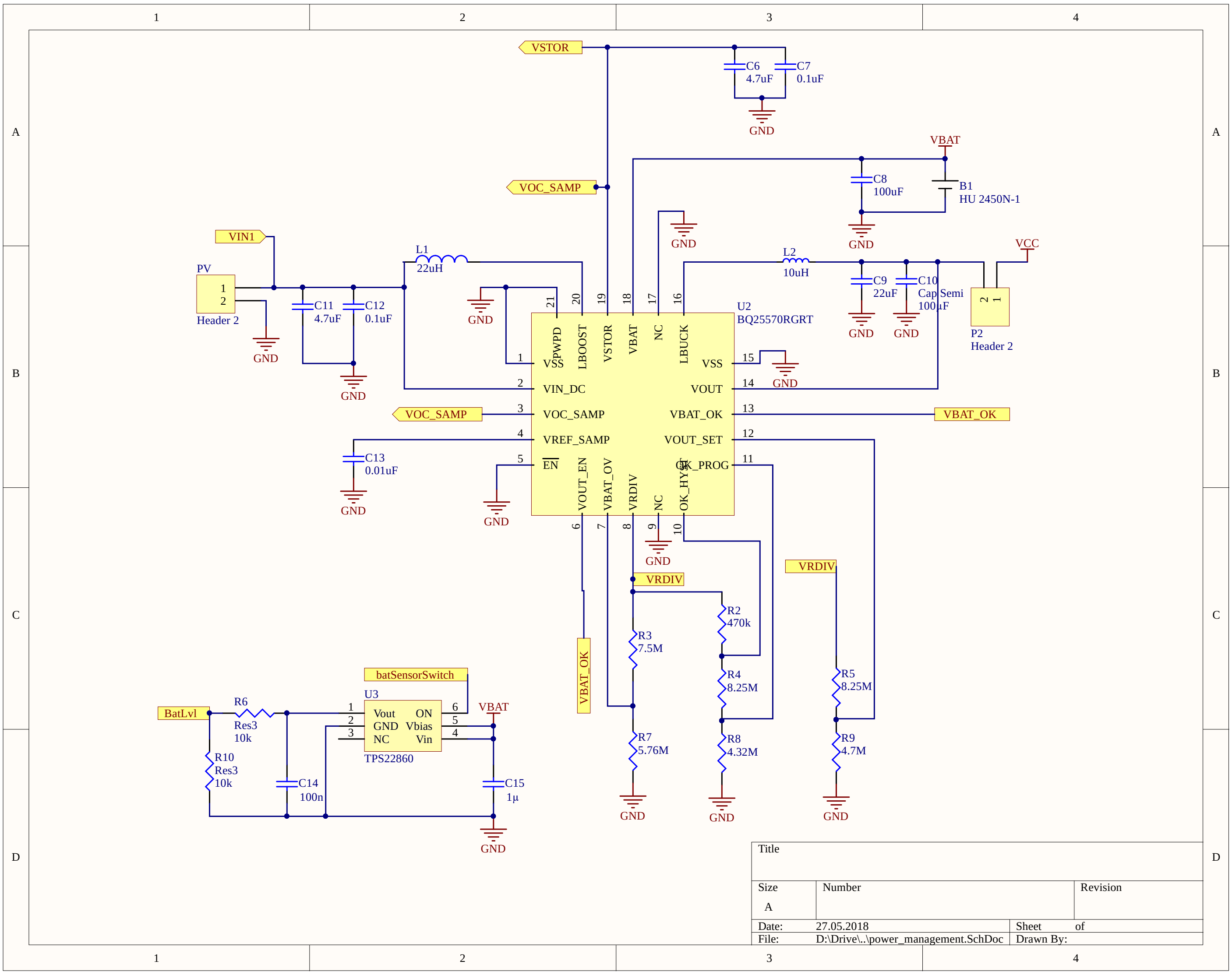
Figure C.6: After reflow soldering

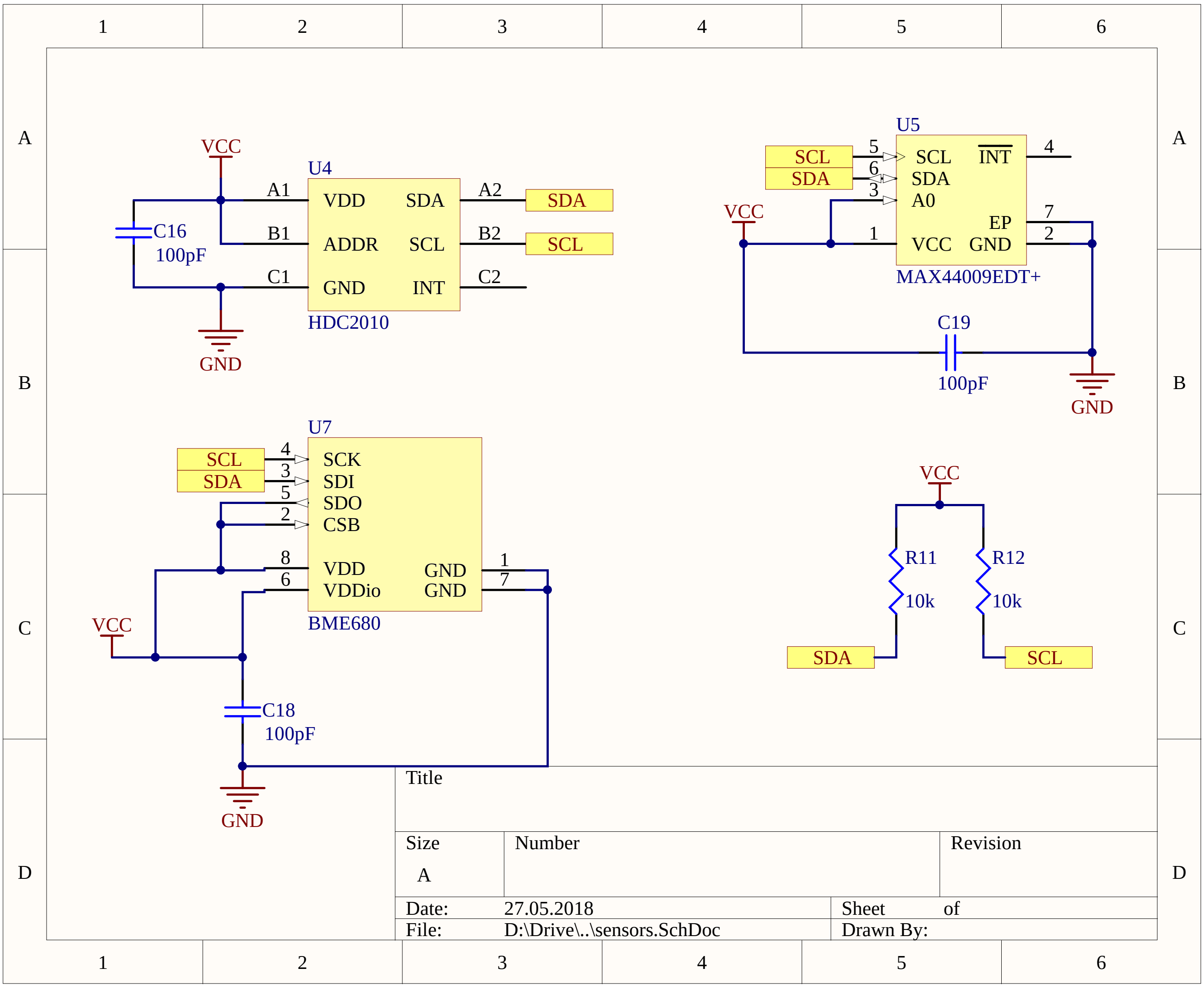
Appendix D

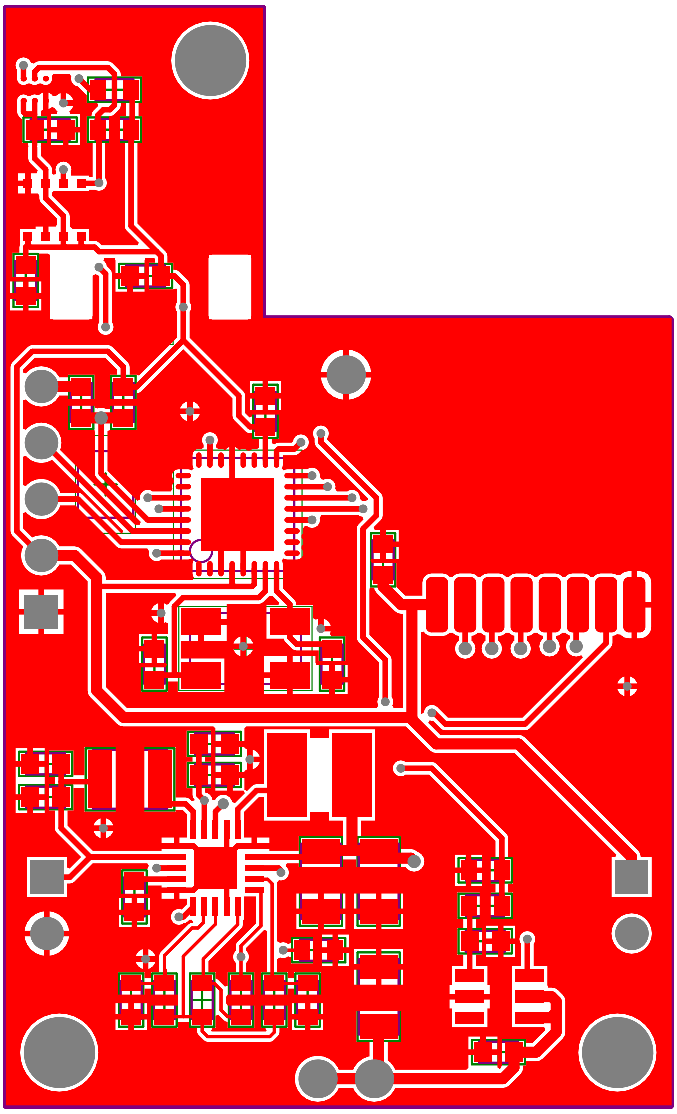
Altium Designs

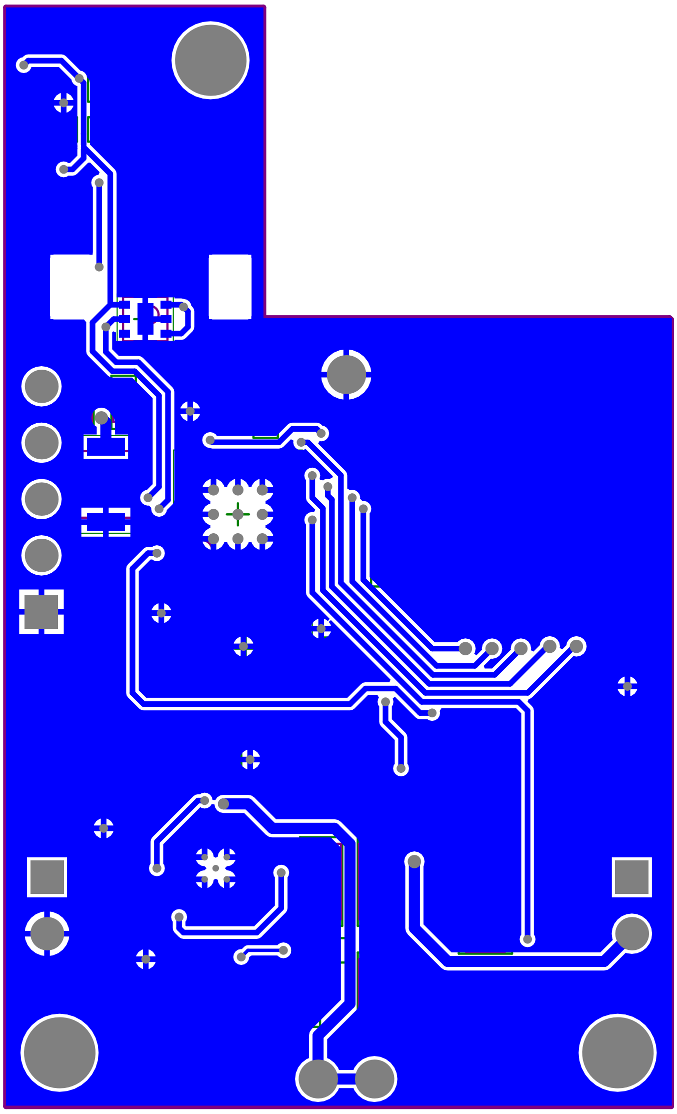
D.1 ATmega328p Node Schematic and PCB

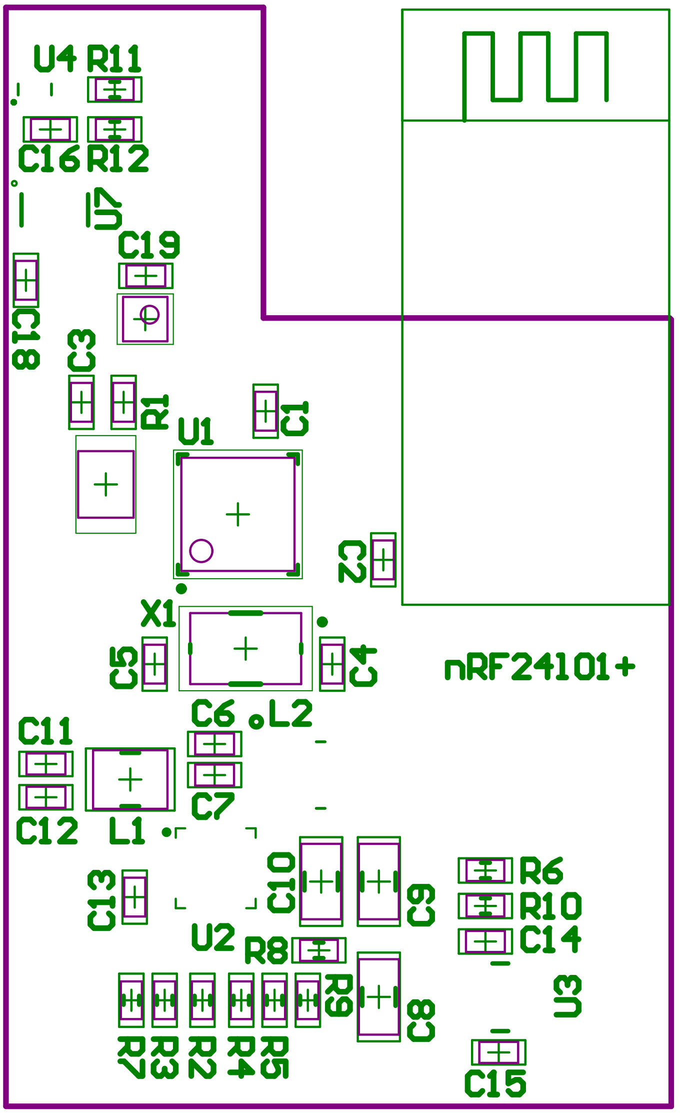

U1
ATmega328P-MU
VCC
VCC
AVCC
AREF
GND
GND
GND
DAP
ADC6
ADC7
PC0 (ADC0/PCINT8)
PC1 (ADC1/PCINT9)
PC2 (ADC2/PCINT10)
PC3 (ADC3/PCINT11)
PC4 (ADC4/SDA/PCINT12)
PC5 (ADC5/SCL/PCINT13)
PC6 (RESET/PCINT14)
PD0 (RXD/PCINT16)
PD1 (TXD/PCINT17)
PD2 (INT0/PCINT18)
PD3 (PCINT19/OC2B/INT1)
PD4 (PCINT20/XCK/T0)
PD5 (PCINT21/OC0B/T1)
PD6 (PCINT22/OC0A/AIN0)
PD7 (PCINT23/AIN1)
PB0 (PCINT0/CLKO/ICP1)
PB1 (PCINT1/OC1A)
PB2 (PCINT2/SS/OC1B)
PB3 (PCINT3/OC2A/MOSI)
PB4 (PCINT4/MISO)
PB5 (SCK/PCINT5)
PB6 (PCINT6/XTAL1/TOSC1)
PB7 (PCINT7/XTAL2/TOSC2)
BatLvl
SDA
SCL
RXI
TXO
D2
batSensorSwitch
D9
D10/SS
D11/MOSI
D12/MISO
D13/SCK
VCC
GND
C1
100pF
R1
10k
C3
100nF
DTR
RST
B3U-1000P
X1
ABM3B-8MHz
C4
20pF
C5
20pF
P1
Header 5
nRF24l01+
VDD
CE
CS
SCK
MOSI
MISO
IRQ
GND
C2
10µF
Title
Size
A
Number
Revision
Date:
27.05.2018
File:
D:\Drive\..\mcu.SchDoc
Sheet
of
Drawn By:

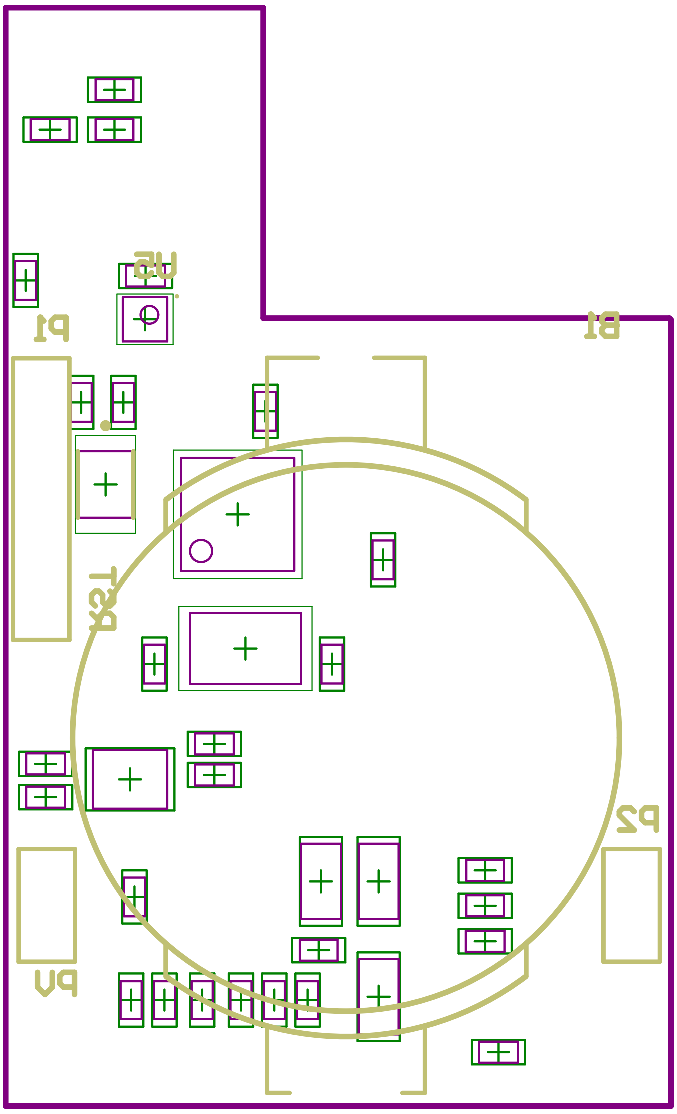

VSTOR
C6
4.7uF
C7
0.1uF
GND
VBAT
C8
100uF
B1
HU 2450N-1
VOC_SAMP
GND
VIN1
L1
22uH
L2
10uH
C9
22uF
C10
Cap Semi
100uF
VCC
PV
1
2
Header 2
C11
4.7uF
C12
0.1uF
GND
GND
21
20
19
18
17
16
VPWPD
LBOOST
VSTOR
VBAT
NC
LBUCK
U2
BQ25570RGRT
P2
Header 2
1
2
VSS
VIN_DC
VOC_SAMP
VREF_SAMP
EN
VSS
VOUT
VBAT_OK
VOUT_SET
OK_PROG
15
14
13
12
11
1
2
3
4
5
VOC_SAMP
VBAT_OK
C13
0.01uF
GND
VOUT_EN
VBAT_OV
VRDIV
NC
OK_HYST
6
7
8
9
10
VRDIV
VRDIV
R2
470k
R3
7.5M
R4
8.25M
R5
8.25M
R7
5.76M
R8
4.32M
R9
4.7M
VBAT_OK
batSensorSwitch
U3
Vout
GND
NC
ON
Vbias
Vin
1
2
3
6
5
4
TPS22860
VBAT
BatLvl
R6
Res3
10k
R10
Res3
10k
C14
100n
C15
1µ
GND
A
B
C
D
1
2
3
4
Title
Size
A
Number
Revision
Date:
27.05.2018
Sheet
of
File:
D:\Drive\..\power_management.SchDoc
Drawn By:


1
2
3
4
5
6
A
B
C
D
VCC
U4
C16
100pF
A1
B1
C1
VDD
ADDR
GND
SDA
SCL
INT
A2
B2
C2
SDA
SCL
HDC2010
GND
U5
SCL
SDA
5
6
3
SCL
SDA
A0
INT
4
EP
7
VCC
1
VCC
GND
2
MAX44009EDT+
C19
100pF
GND
U7
SCL
SDA
4
3
5
2
SCK
SDI
SDO
CSB
8
6
VDD
VDDio
GND
GND
1
7
BME680
VCC
C18
100pF
GND
VCC
R11
10k
R12
10k
SDA
SCL
Title
Size
A
Number
Revision
Date:
27.05.2018
Sheet
of
File:
D:\Drive\..\sensors.SchDoc
Drawn By:

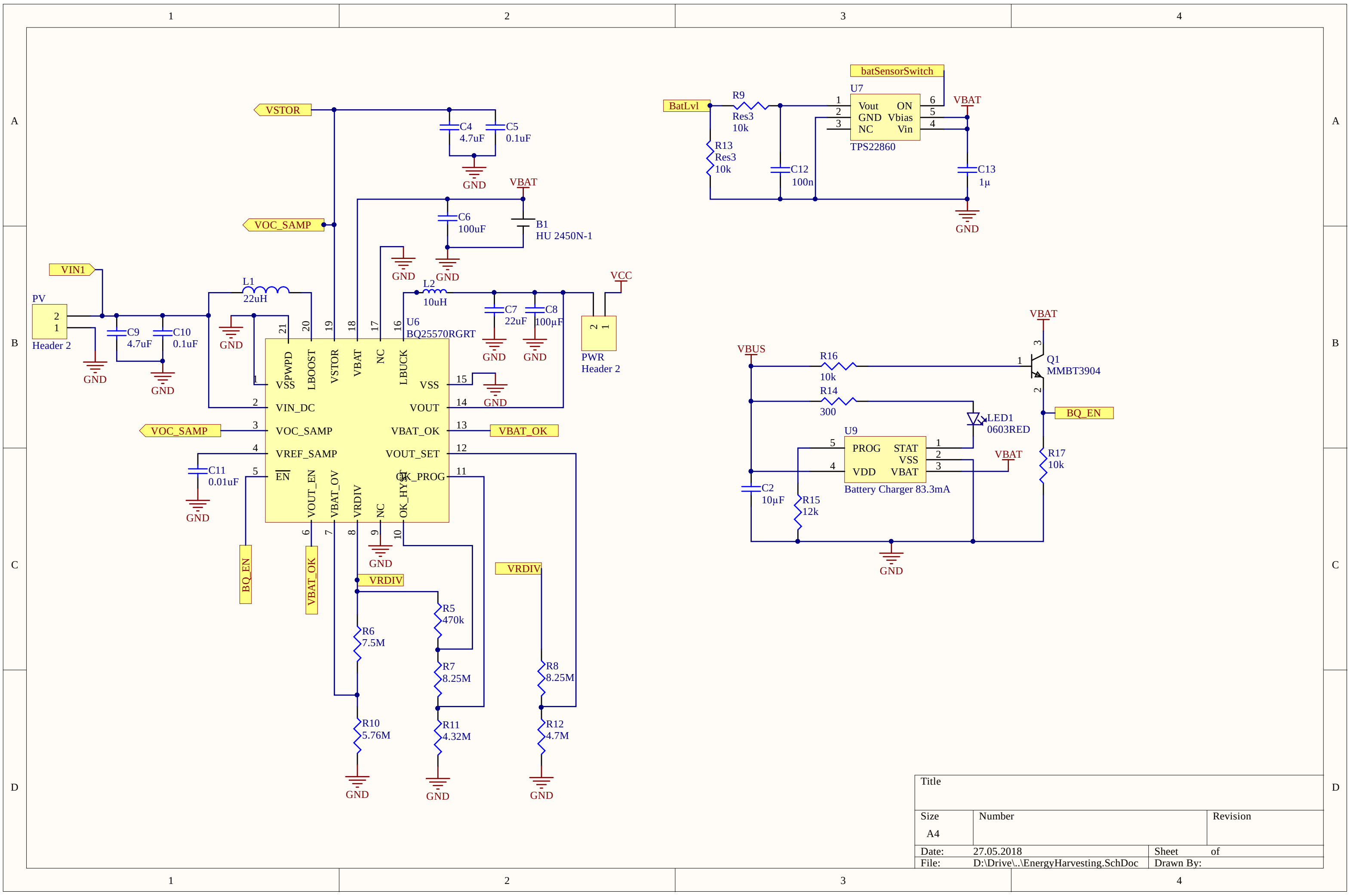




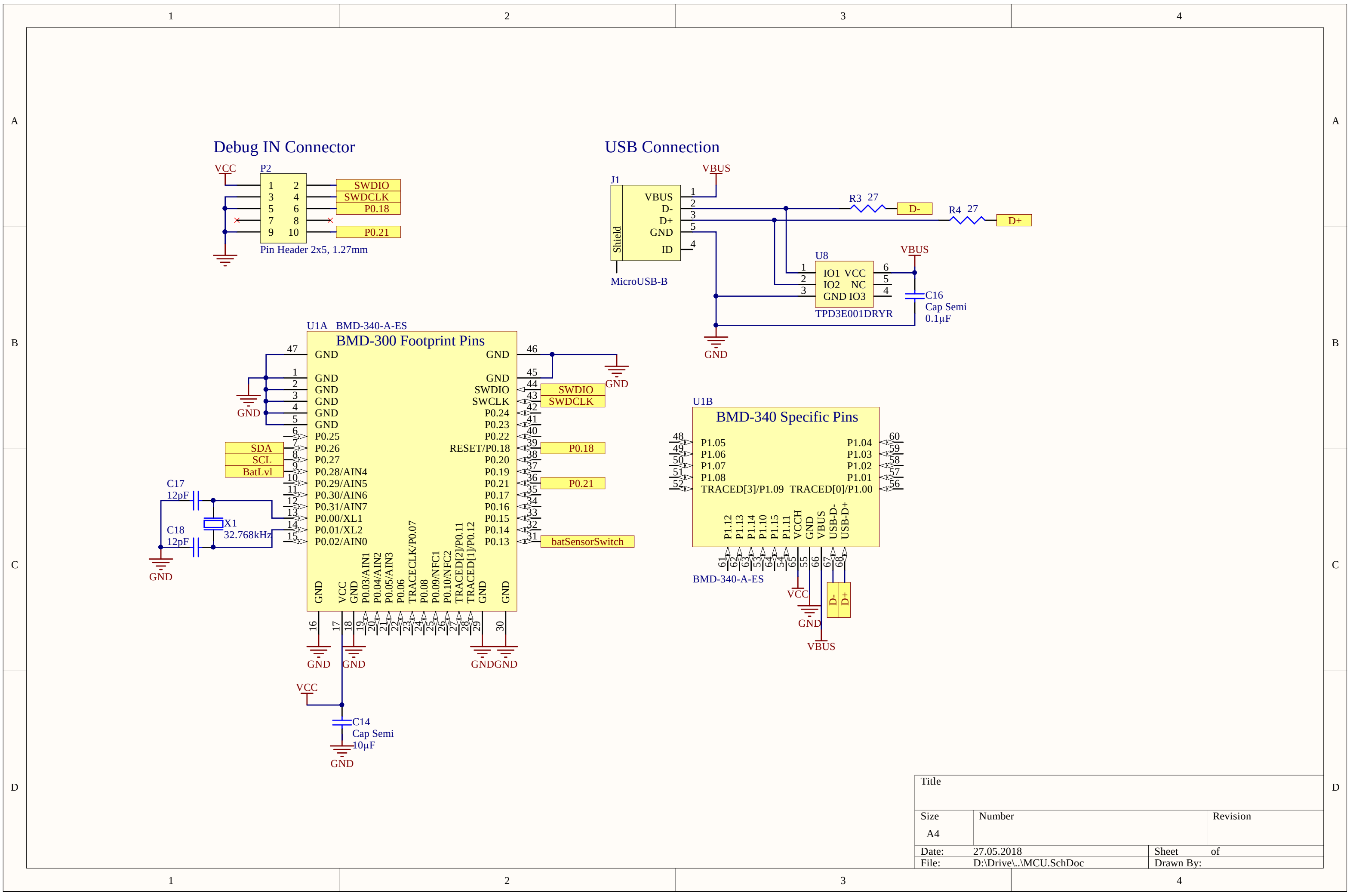

U4
R11
C16
R12
U7
C19
C18
C3
R1
U1
C1
C2
X1
C5
C4
nRF24l01+
C6
L2
C11
C12
L1
C7
C13
C10
C9
R6
R10
C14
U2
R8
R9
U3
C8
R7
R3
R2
R4
R5
C15

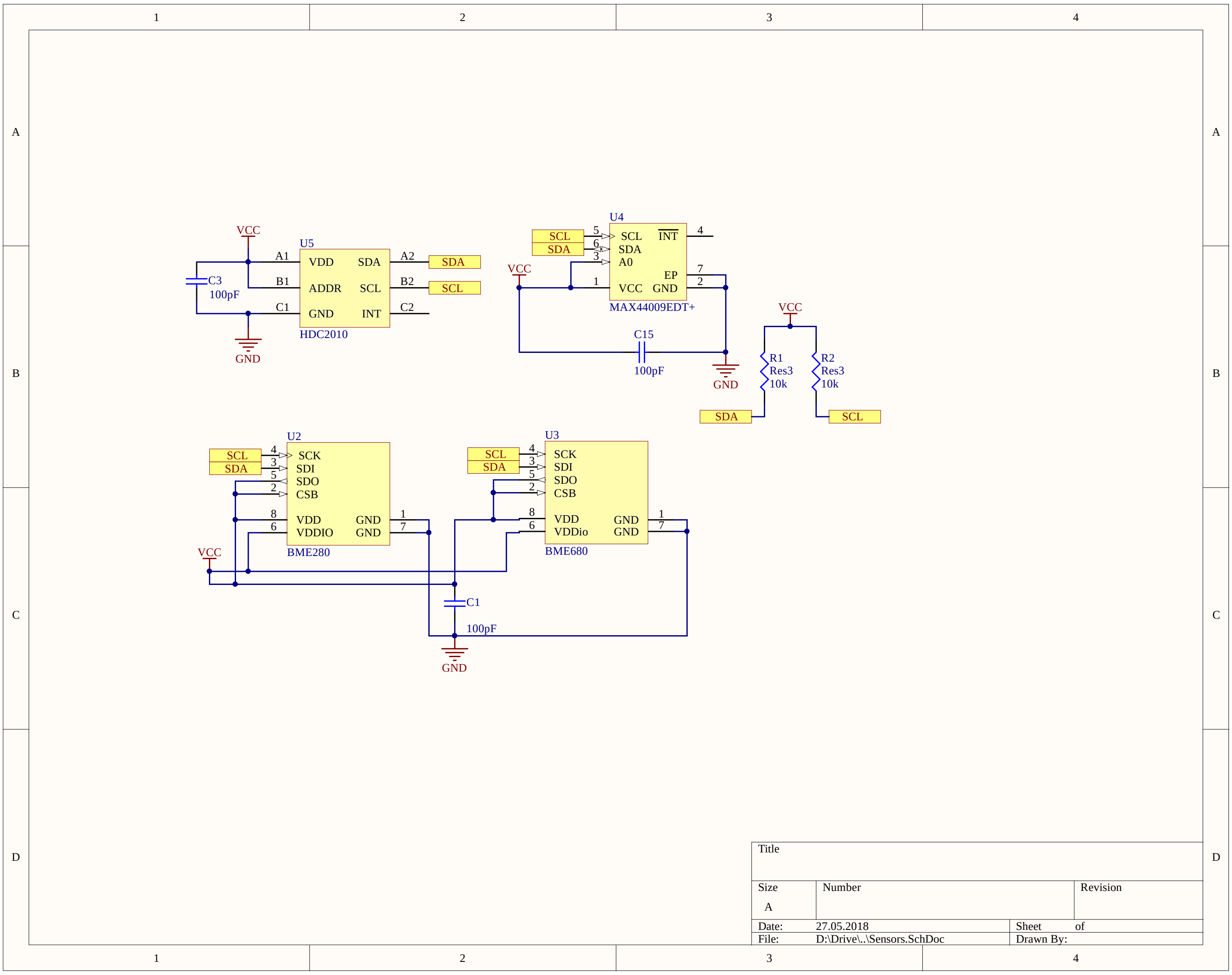

U5
P1
B1
P2
PV

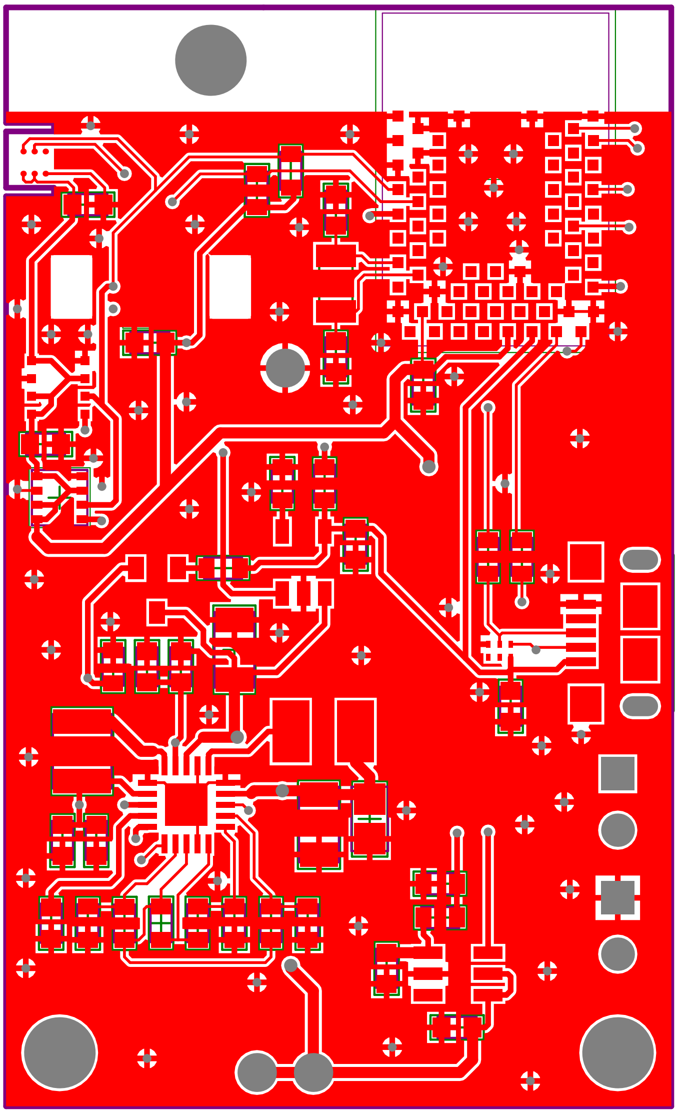
D.2 nRF52840 Node Schematic and PCB

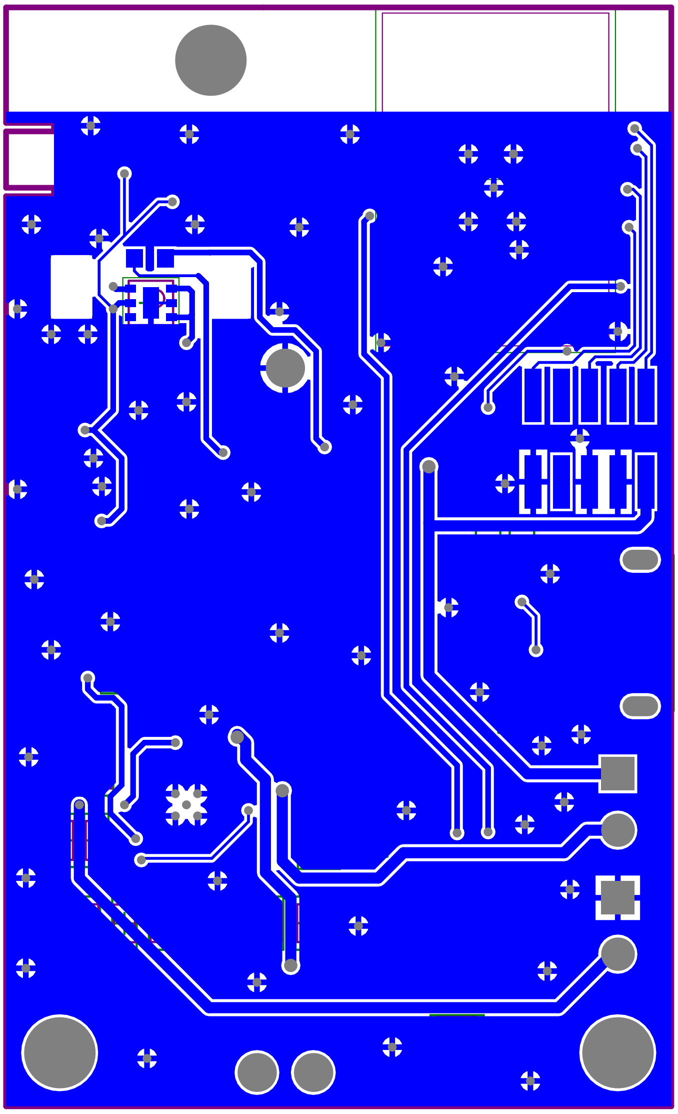


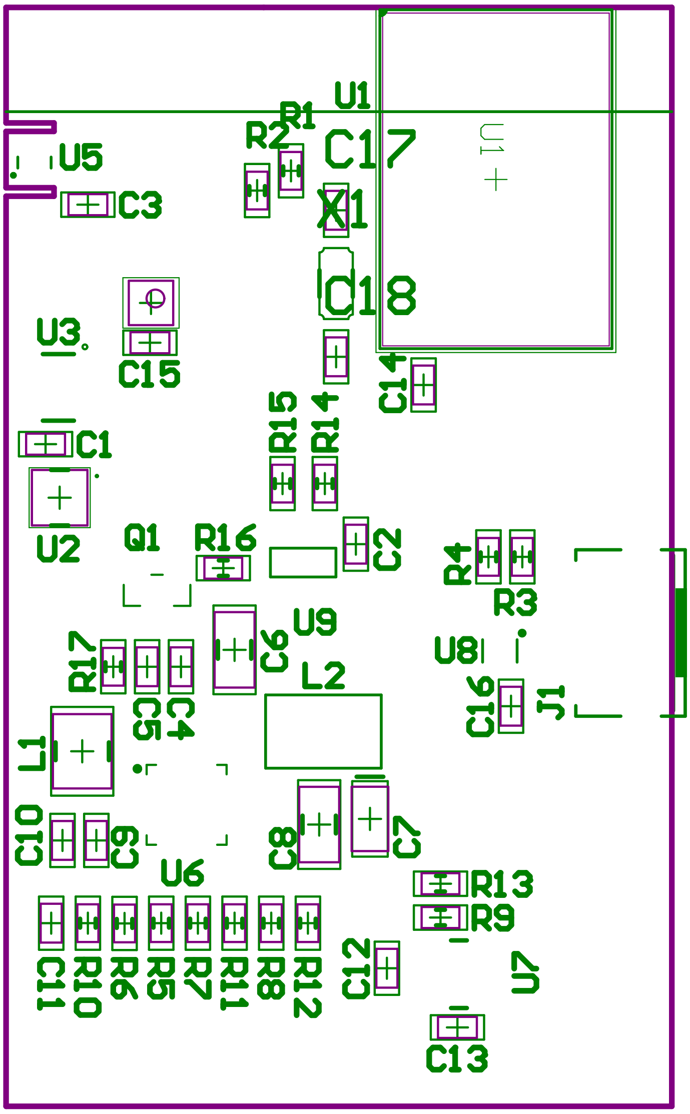
VSTOR
C4
4.7uF
C5
0.1uF
GND
VBAT
C6
100uF
B1
HU 2450N-1
VOC_SAMP
VIN1
PV
Header 2
C9
4.7uF
C10
0.1uF
L1
22uH
L2
10uH
C7
22uF
C8
100µF
VCC
PWR
Header 2
U6
BQ25570RGRT
VSS
VIN_DC
VOC_SAMP
VREF_SAMP
EN
VPWPD
LBOOST
VSTOR
VBAT
NC
LBUCK
VOUT
VBAT_OK
VOUT_SET
OK_PROG
VOUT_EN
VBAT_OV
VRDIV
OK_HYST
C11
0.01uF
BQ_EN
VBAT_OK
VRDIV
R5
470k
R6
7.5M
R7
8.25M
R8
8.25M
R10
5.76M
R11
4.32M
R12
4.7M
batSensorSwitch
BatLvl
R9
Res3
10k
R13
Res3
10k
U7
Vout
GND
NC
ON
Vbias
Vin
TPS22860
C12
100n
C13
1µ
VBUS
R16
10k
R14
300
LED1
0603RED
Q1
MMBT3904
BQ_EN
R17
10k
U9
PROG
VDD
STAT
VSS
VBAT
Battery Charger 83.3mA
C2
10µF
R15
12k
Title
Size
A4
Number
Revision
Date:
27.05.2018
File:
D:\Drive\..\EnergyHarvesting.SchDoc
Sheet
of
Drawn By:

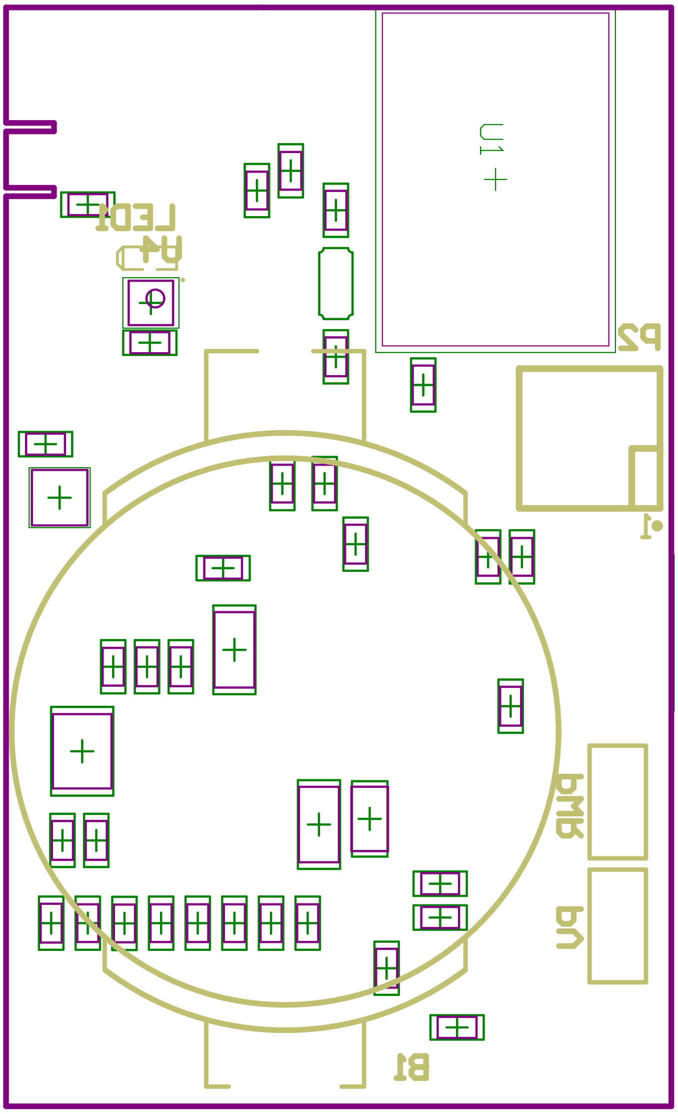

Debug IN Connector
P2
VCC
SWDIO
SWDCLK
P0.18
P0.21
Pin Header 2x5, 1.27mm
USB Connection
J1
VBUS
D-
D+
GND
ID
Shield
MicroUSB-B
R3 27
R4 27
U8
IO1 VCC
IO2 NC
GND IO3
TPD3E001DRYR
C16
Cap Semi
0.1µF
U1A BMD-340-A-ES
BMD-300 Footprint Pins
GND
SWDIO
SWCLK
P0.24
P0.23
P0.22
RESET/P0.18
P0.20
P0.19
P0.21
P0.17
P0.16
P0.15
P0.14
P0.13
P0.25
P0.26
P0.27
P0.28/AIN4
P0.29/AIN5
P0.30/AIN6
P0.31/AIN7
P0.00/XL1
P0.01/XL2
P0.02/AIN0
SDA
SCL
BatLvl
batSensorSwitch
C17 12pF
C18 12pF
X1 32.768kHz
VCC
P0.03/AIN1
P0.04/AIN2
P0.05/AIN3
P0.06
TRACECLK/P0.07
P0.08
P0.09/NFC1
P0.10/NFC2
TRACED[2]/P0.11
TRACED[1]/P0.12
C14
Cap Semi
10µF
U1B
BMD-340 Specific Pins
P1.05
P1.06
P1.07
P1.08
TRACED[3]/P1.09
P1.04
P1.03
P1.02
P1.01
TRACED[0]/P1.00
P1.12
P1.13
P1.14
P1.10
P1.15
P1.11
VCCH
USB-D-
USB-D+
BMD-340-A-ES
Title
Size
A4
Number
Revision
Date: 27.05.2018
Sheet of
File: D:\Drive\..\MCU.SchDoc
Drawn By:
A
B
C
D
1
2
3
4


1
2
3
4
A
B
C
D
VCC
U5
A1
VDD
SDA
A2
SDA
C3
100pF
B1
ADDR
SCL
B2
SCL
C1
GND
INT
C2
HDC2010
GND
U4
SCL
SDA
5
6
3
SCL
SDA
A0
INT
4
VCC
EP
7
1
VCC
GND
2
MAX44009EDT+
C15
100pF
GND
VCC
R1
Res3
10k
R2
Res3
10k
SDA
SCL
U2
SCL
SDA
4
3
5
2
SCK
SDI
SDO
CSB
8
6
VDD
VDDIO
GND
GND
1
7
BME280
VCC
U3
SCL
SDA
4
3
5
2
SCK
SDI
SDO
CSB
8
6
VDD
VDDio
GND
GND
1
7
BME680
C1
100pF
GND
Title
Size
A
Number
Revision
Date:
27.05.2018
Sheet
of
File:
D:\Drive\..\Sensors.SchDoc
Drawn By:

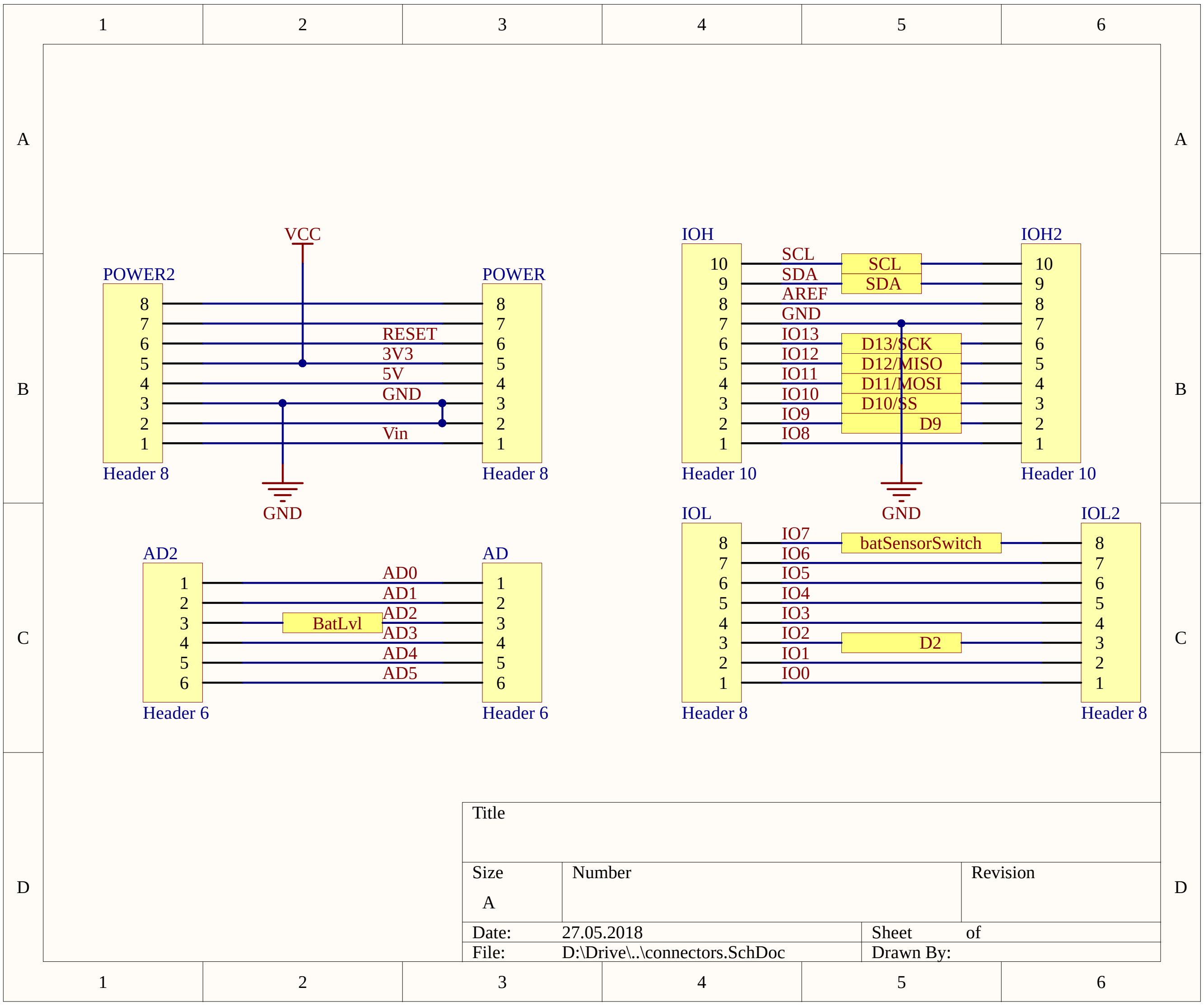


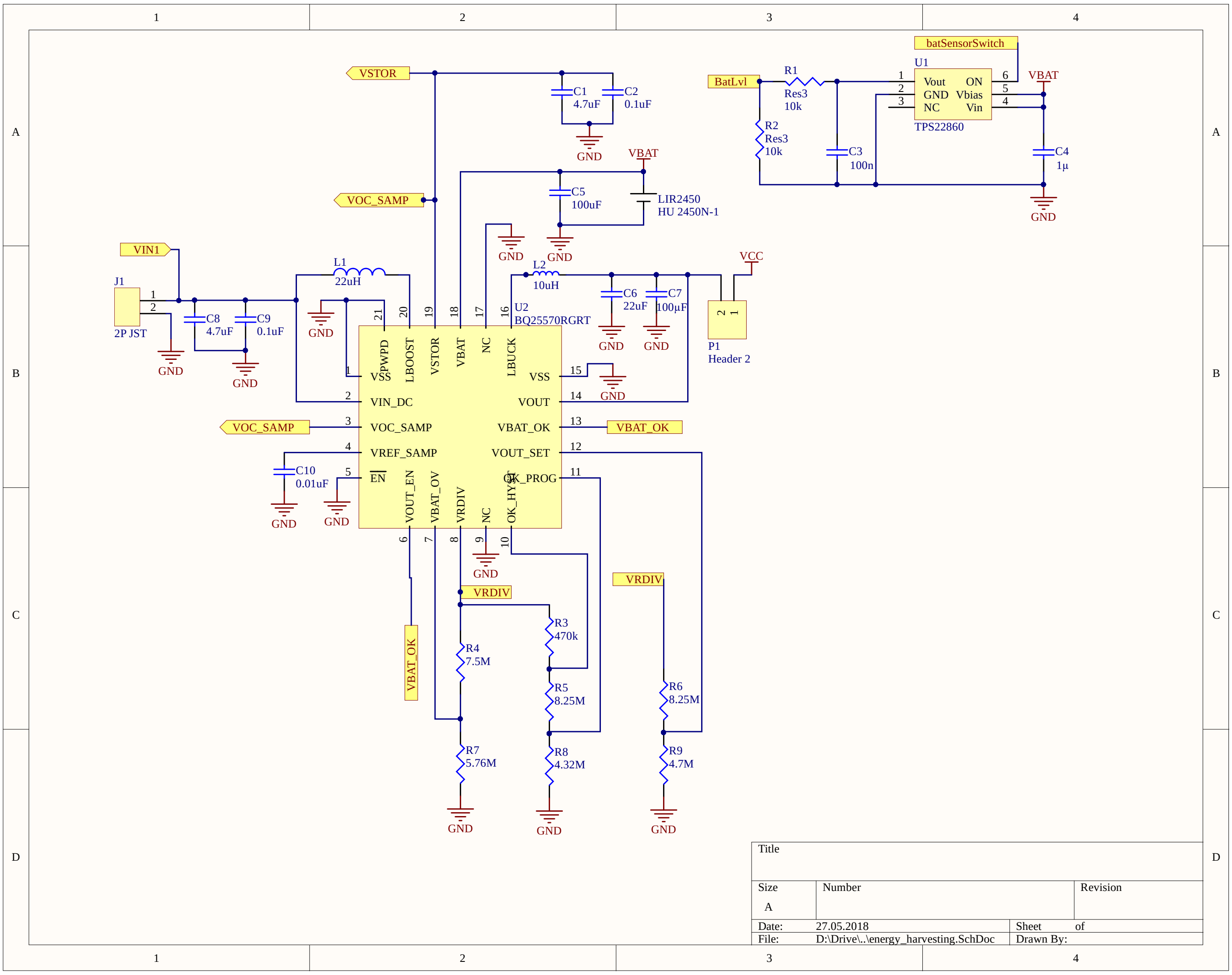



U1
R1
R2
C17
U5
C3
X1
C18
U3
C15
C14
R15
R14
C1
U2
Q1
R16
C2
R4
R3
U9
U8
R17
C6
L2
C16
J1
C5
C4
L1
C10
C9
C8
C7
U6
R13
R9
C11
R10
R6
R5
R7
R11
R8
R12
C12
U7
C13
U1

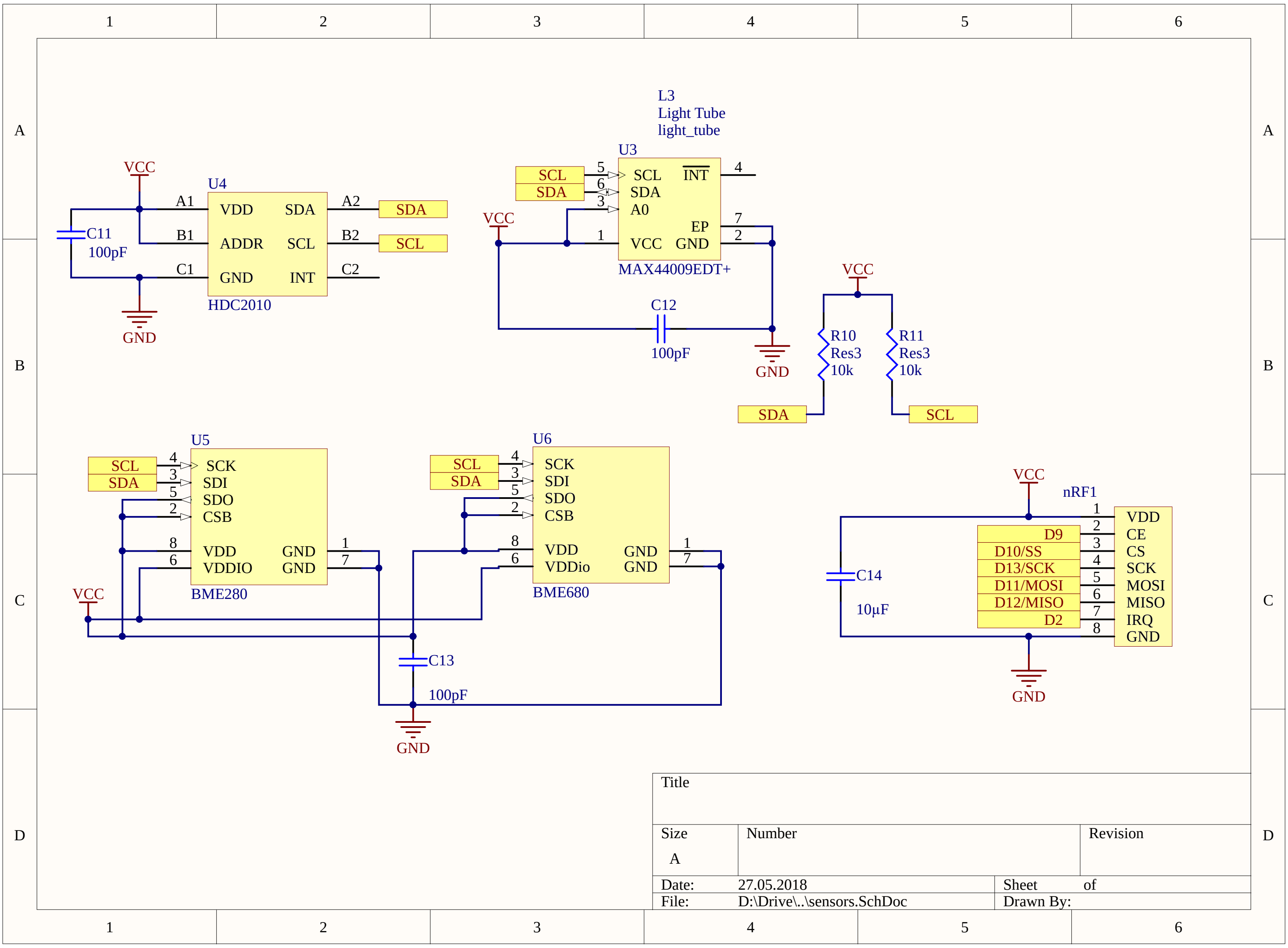

U1
LED1
R4
P2
1
pwr
up
B1

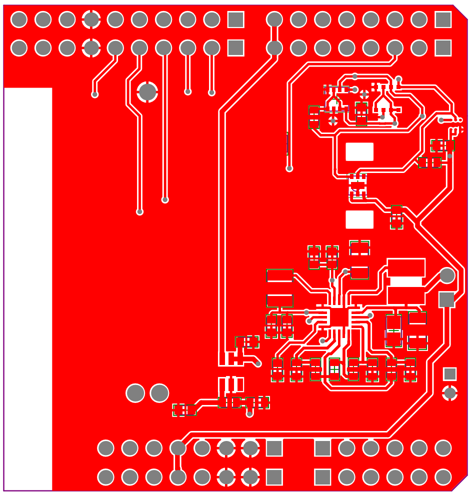
D.3 nRF52840 PDK Brekout Board Schematic and PCB

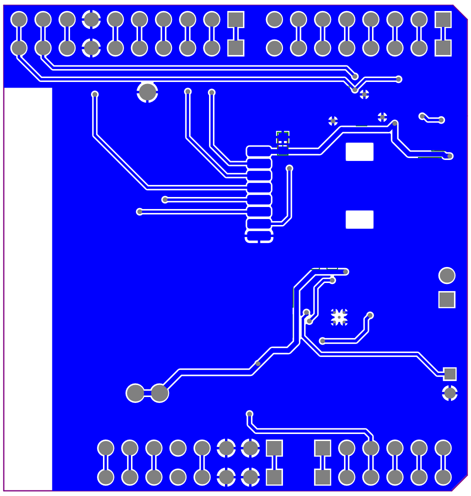

VCC
POWER2
POWER
Header 8
Header 8
RESET
3V3
5V
GND
Vin
GND
IOH
IOH2
Header 10
Header 10
SCL
SDA
AREF
GND
IO13
IO12
IO11
IO10
IO9
IO8
D13/SCK
D12/MISO
D11/MOSI
D10/SS
D9
GND
AD2
AD
Header 6
Header 6
AD0
AD1
AD2
AD3
AD4
AD5
BatLvl
IOL
IOL2
Header 8
Header 8
IO7
IO6
IO5
IO4
IO3
IO2
IO1
IO0
batSensorSwitch
D2
Title
Size
A
Number
Revision
Date:
27.05.2018
File:
D:\Drive\..\connectors.SchDoc
Sheet
of
Drawn By:

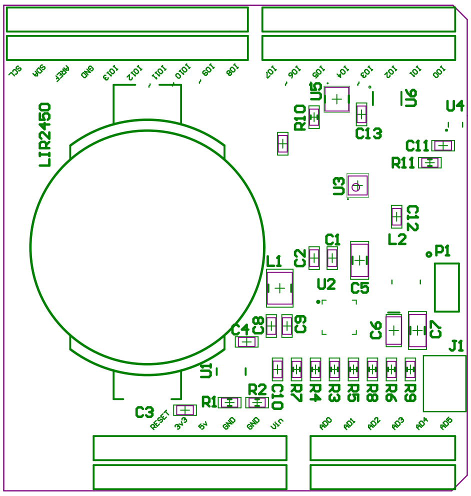


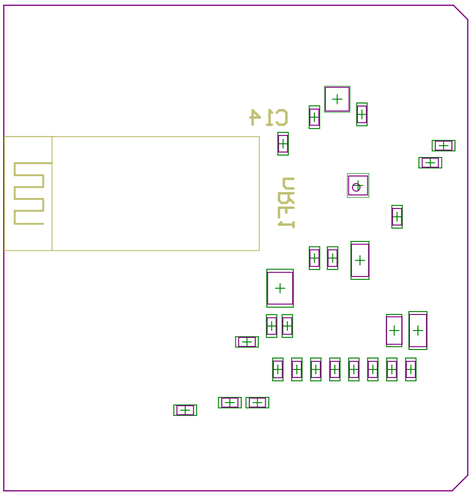
1
2
3
4
A
B
C
D
VSTOR
C1
4.7uF
C2
0.1uF
GND
VBAT
VOC_SAMP
C5
100uF
LIR2450
HU 2450N-1
batSensorSwitch
U1
BatLvl
R1
Res3
10k
R2
Res3
10k
C3
100n
Vout
GND
NC
ON
Vbias
Vin
TPS22860
C4
1µ
VIN1
J1
2P JST
C8
4.7uF
C9
0.1uF
L1
22uH
L2
10uH
C6
22uF
C7
100µF
VCC
P1
Header 2
U2
BQ25570RGRT
VSS
VIN_DC
VOC_SAMP
VREF_SAMP
EN
VPWPD
LBOOST
VSTOR
VBAT
NC
LBUCK
VOUT
VBAT_OK
VOUT_SET
OK_PROG
VOUT_EN
VBAT_OV
VRDIV
OK_HYST
C10
0.01uF
VBAT_OK
VRDIV
R3
470k
R4
7.5M
R5
8.25M
R6
8.25M
R7
5.76M
R8
4.32M
R9
4.7M
Title
Size
A
Number
Revision
Date:
27.05.2018
Sheet
of
File:
D:\Drive\..\energy_harvesting.SchDoc
Drawn By:


1
2
3
4
5
6
A
B
C
D
L3
Light Tube
light_tube
U3
SCL
SDA
5
6
3
SCL
SDA
A0
INT
4
VCC
1
VCC
EP
7
GND
2
MAX44009EDT+
C12
100pF
GND
VCC
U4
A1
VDD
SDA
A2
SDA
C11
100pF
B1
ADDR
SCL
B2
SCL
C1
GND
INT
C2
GND
HDC2010
VCC
R10
Res3
10k
R11
Res3
10k
SDA
SCL
U5
SCL
SDA
4
3
5
2
SCK
SDI
SDO
CSB
8
6
VDD
VDDIO
GND
GND
1
7
BME280
VCC
U6
SCL
SDA
4
3
5
2
SCK
SDI
SDO
CSB
8
6
VDD
VDDio
GND
GND
1
7
BME680
C13
100pF
GND
VCC
nRF1
C14
10µF
D9
D10/SS
D13/SCK
D11/MOSI
D12/MISO
D2
1
2
3
4
5
6
7
8
VDD
CE
CS
SCK
MOSI
MISO
IRQ
GND
GND
Title
Size
A
Number
Revision
Date:
27.05.2018
Sheet
of
File:
D:\Drive\..\sensors.SchDoc
Drawn By:

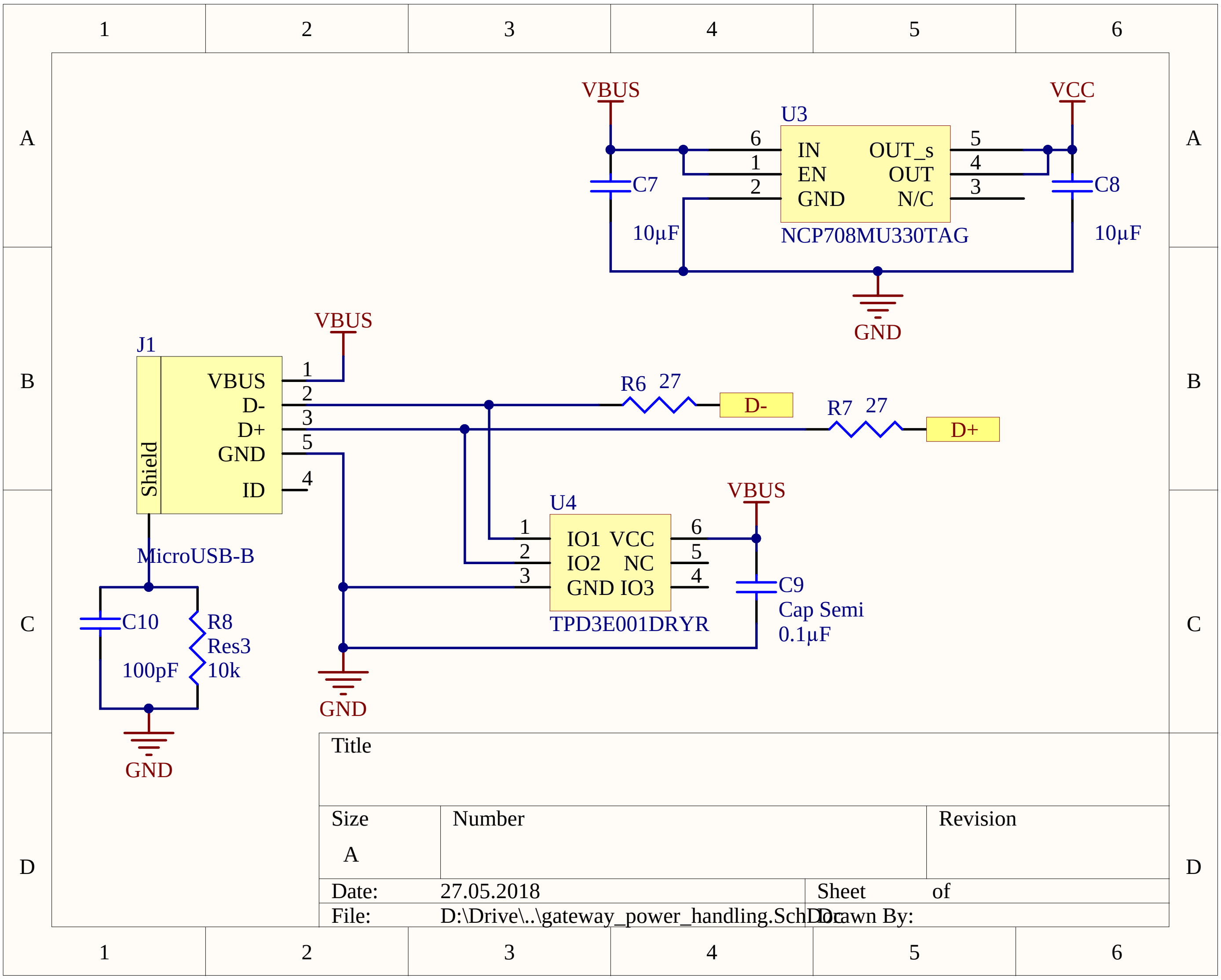


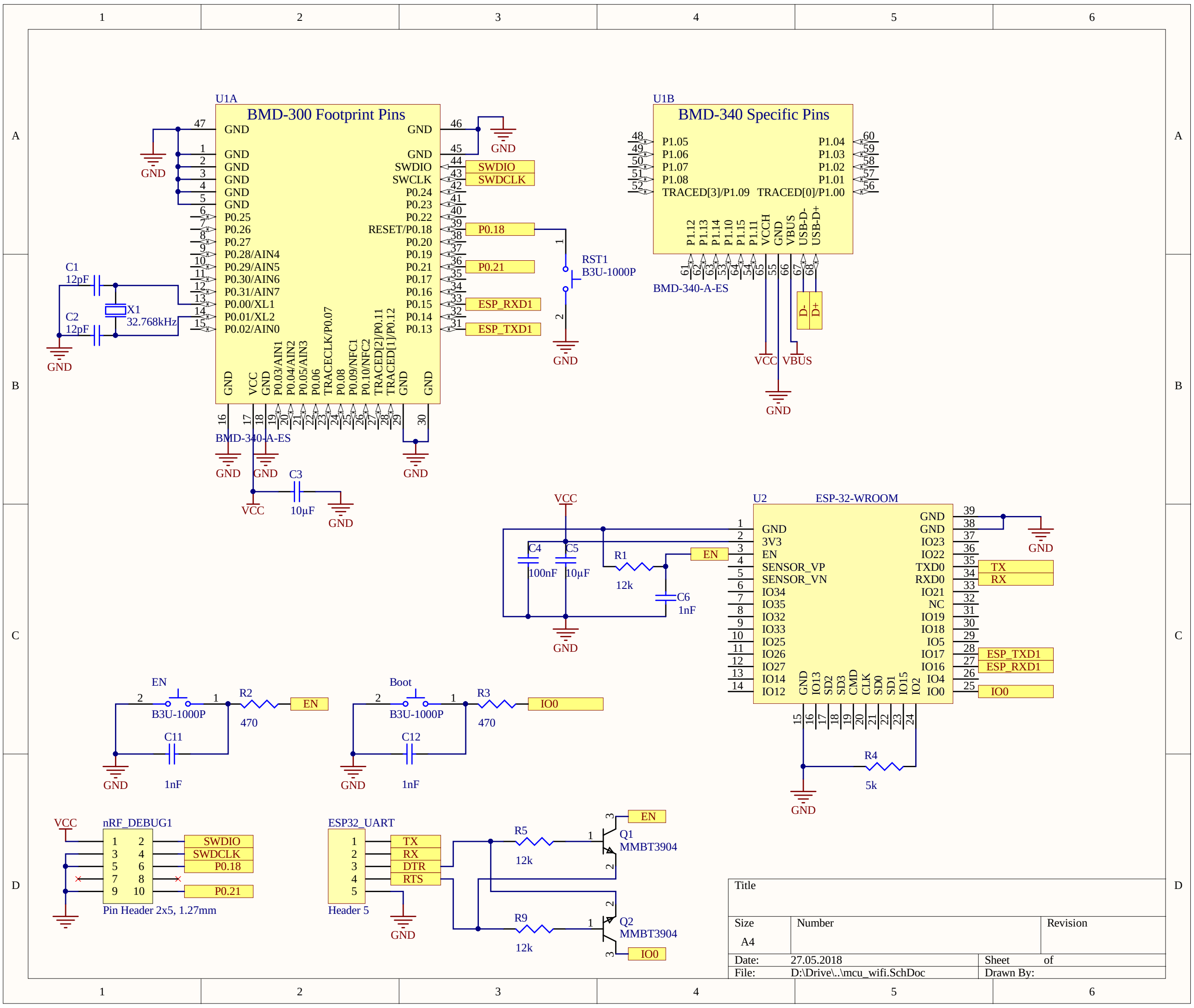


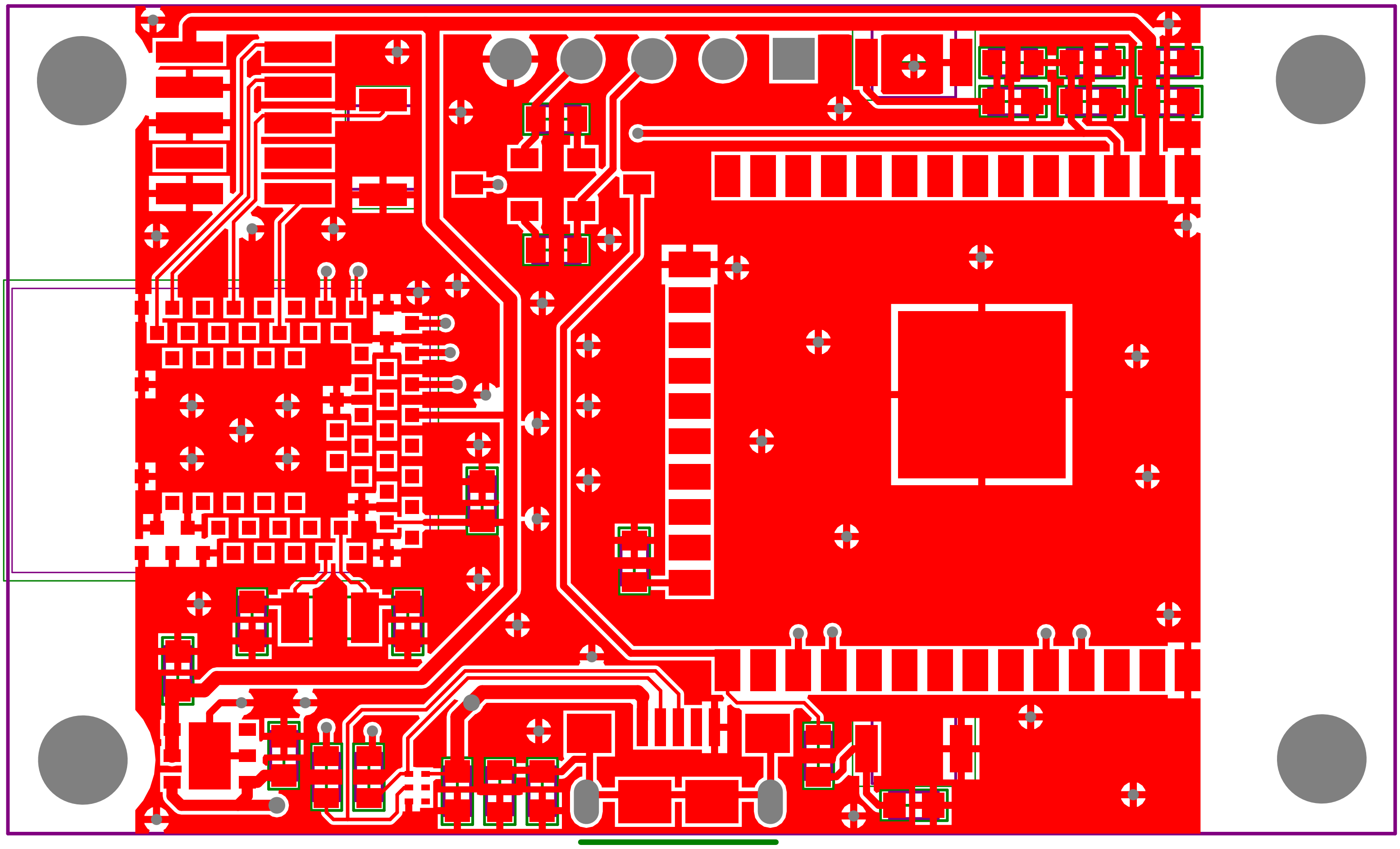

SCL
SDA
AREF
GND
IO13
IO12
IO11
IO10
IO9
IO8
IO7
IO6
IO5
IO4
IO3
IO2
IO1
IO0
LIR2450
U5
R10
C13
U6
U4
C11
R11
U3
C12
L2
P1
C1
C2
L1
U2
C5
C4
C8
C9
C6
C7
J1
U1
C10
R7
R4
R3
R5
R8
R6
R9
R2
R1
C3
RESET
3v3
5v
GND
GND
Vin
AD0
AD1
AD2
AD3
AD4
AD5

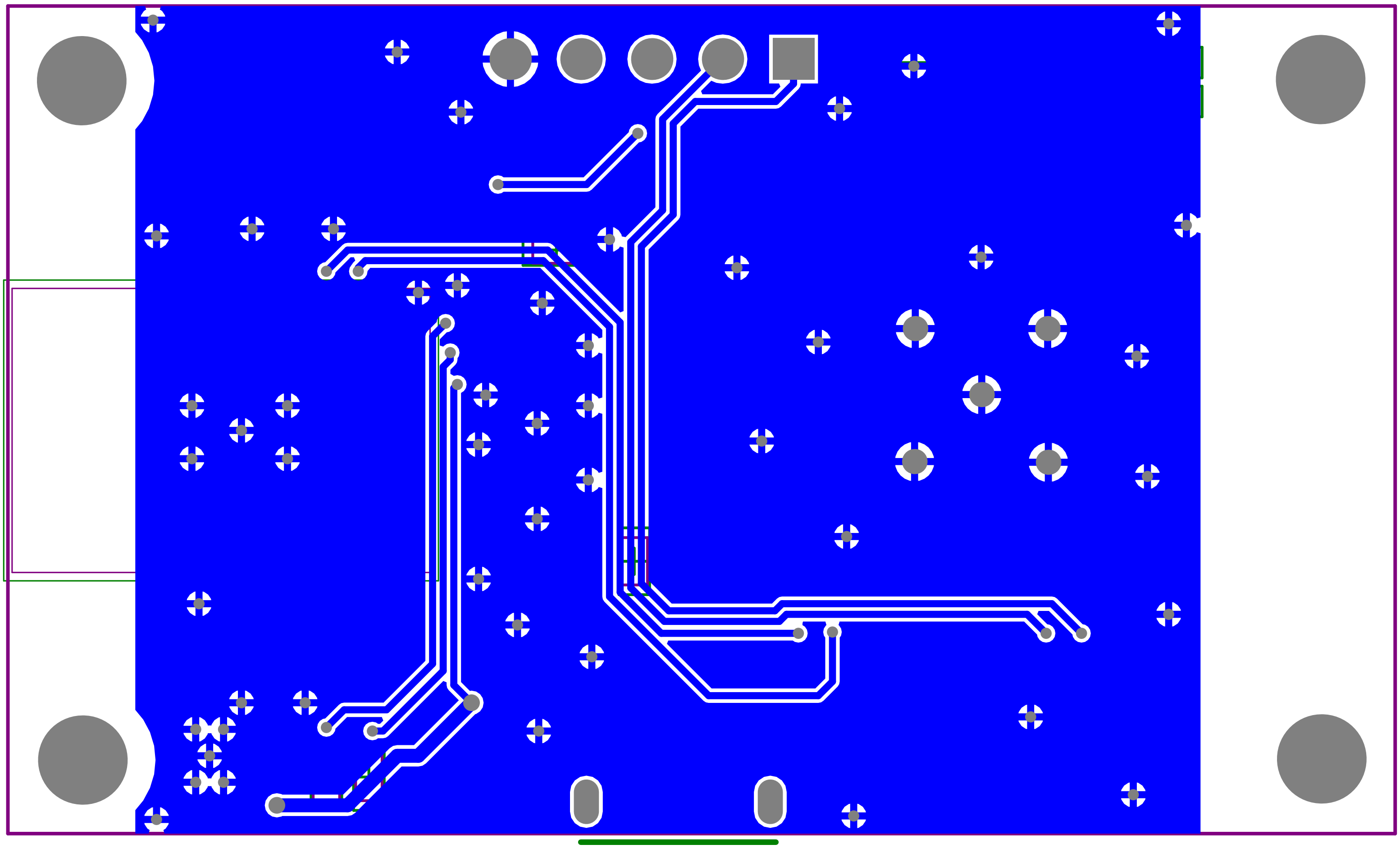

C14

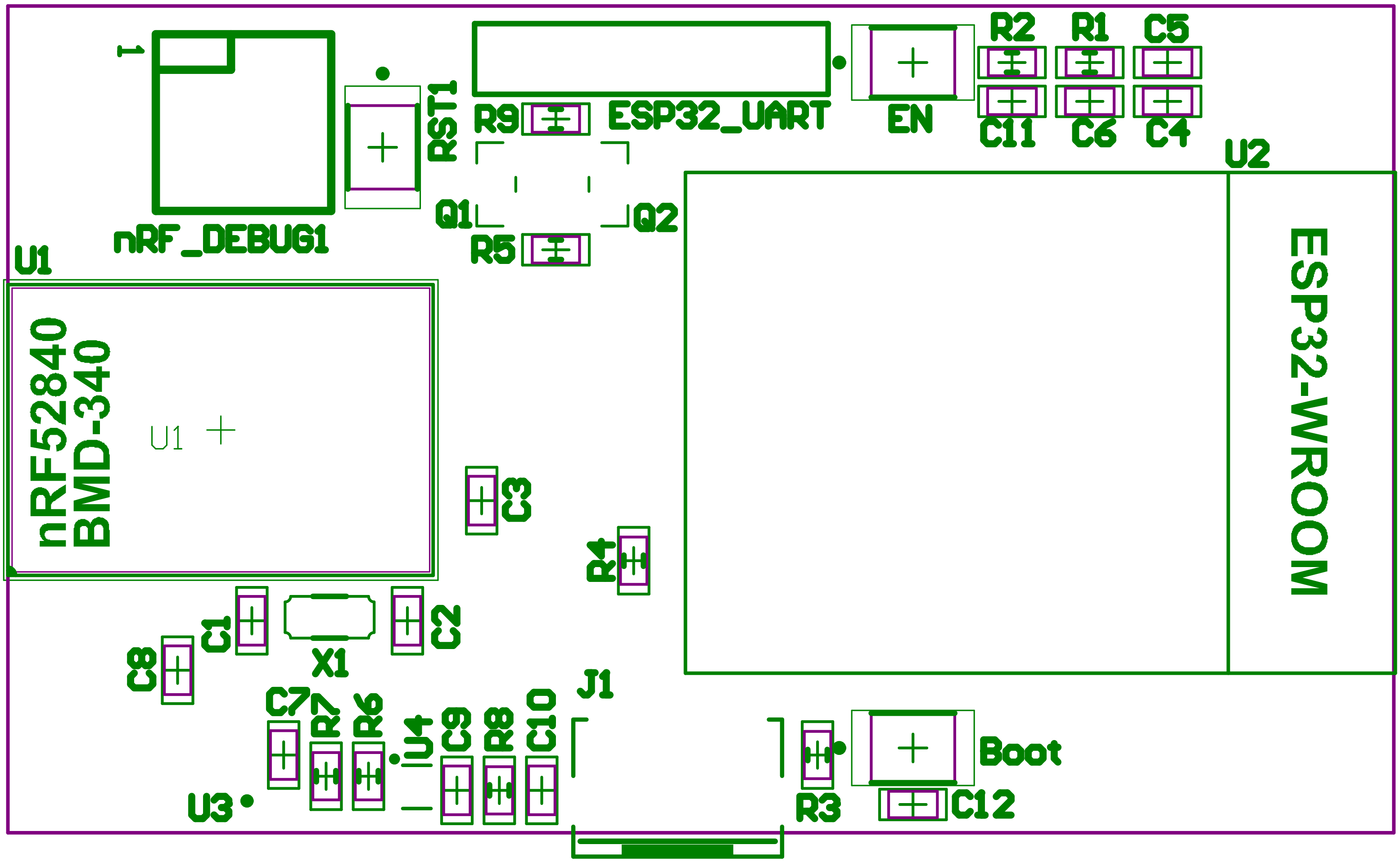
D.4 nRF52840 Gateway Schematic and PCB

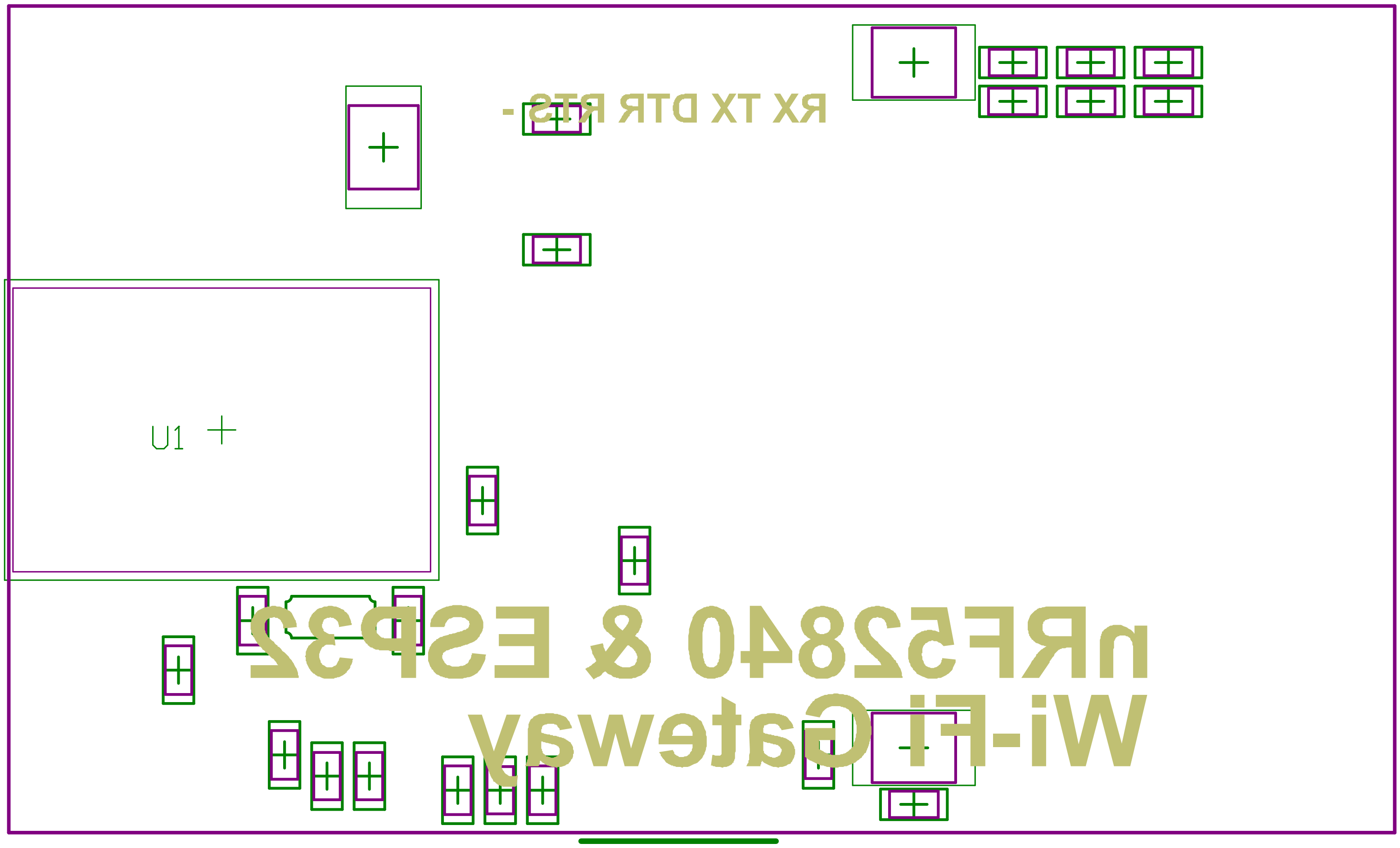

1
2
3
4
5
6
A
B
C
D
VBUS
VCC
U3
IN
EN
GND
OUT_s
OUT
N/C
6
1
2
5
4
3
C7
10µF
C8
10µF
NCP708MU330TAG
GND
VBUS
J1
VBUS
D-
D+
GND
ID
Shield
1
2
3
5
4
MicroUSB-B
R6
27
D-
R7
27
D+
U4
IO1
IO2
GND
VCC
NC
IO3
1
2
3
6
5
4
VBUS
C9
Cap Semi
0.1µF
TPD3E001DRYR
C10
100pF
R8
Res3
10k
GND
GND
Title
Size
A
Number
Revision
Date:
27.05.2018
Sheet
of
File:
D:\Drive\..\gateway_power_handling.SchDoc
Drawn By:


U1A
BMD-300 Footprint Pins
GND
SWDIO
SWCLK
P0.24
P0.23
P0.22
RESET/P0.18
P0.20
P0.19
P0.21
P0.17
P0.16
P0.15
P0.14
P0.13
P0.25
P0.26
P0.27
P0.28/AIN4
P0.29/AIN5
P0.30/AIN6
P0.31/AIN7
P0.00/XL1
P0.01/XL2
P0.02/AIN0
VCC
P0.03/AIN1
P0.04/AIN2
P0.05/AIN3
P0.06
TRACECLK/P0.07
P0.08
P0.09/NFC1
P0.10/NFC2
TRACED[2]/P0.11
TRACED[1]/P0.12
BMD-340-A-ES
C1
12pF
C2
12pF
X1
32.768kHz
C3
10µF
RST1
B3U-1000P
U1B
BMD-340 Specific Pins
P1.05
P1.06
P1.07
P1.08
TRACED[3]/P1.09
P1.04
P1.03
P1.02
P1.01
TRACED[0]/P1.00
P1.12
P1.13
P1.14
P1.10
P1.15
P1.11
VCCH
VBUS
USB-D-
USB-D+
D-
D+
VCC
VBUS
U2
ESP-32-WROOM
3V3
EN
SENSOR_VP
SENSOR_VN
IO34
IO35
IO32
IO33
IO25
IO26
IO27
IO14
IO12
IO13
SD2
SD3
CMD
CLK
SD0
SD1
IO15
IO2
IO23
IO22
TXD0
RXD0
IO21
NC
IO19
IO18
IO5
IO17
IO16
IO4
IO0
TX
RX
ESP_TXD1
ESP_RXD1
P0.18
P0.21
C4
100nF
C5
10µF
R1
12k
C6
1nF
R2
470
R3
470
C11
1nF
C12
1nF
Boot
R4
5k
nRF_DEBUG1
Pin Header 2x5, 1.27mm
ESP32_UART
Header 5
DTR
RTS
R5
12k
R9
12k
Q1
MMBT3904
Q2
MMBT3904
Title
Size
A4
Number
Revision
Date: 27.05.2018
File: D:\Drive\..\mcu_wifi.SchDoc
Sheet of
Drawn By:






1
RST1
R9
ESP32_UART
EN
R2
R1
C5
C11
C6
C4
U2
nRF_DEBUG1
Q1
Q2
R5
U1
nRF52840
BMD-340
U1
ESP32-WROOM
C3
R4
C1
C2
X1
C8
C7
R7
R6
U4
C9
R8
C10
J1
Boot
U3
R3
C12


RX TX DTR RTS -
U1
nRF52840 & ESP32
Wi-Fi Gateway

UiA University of Agder
Master's thesis
Faculty of Engineering and Science
Department of ICT