Bachelor's thesis

Annik Riise

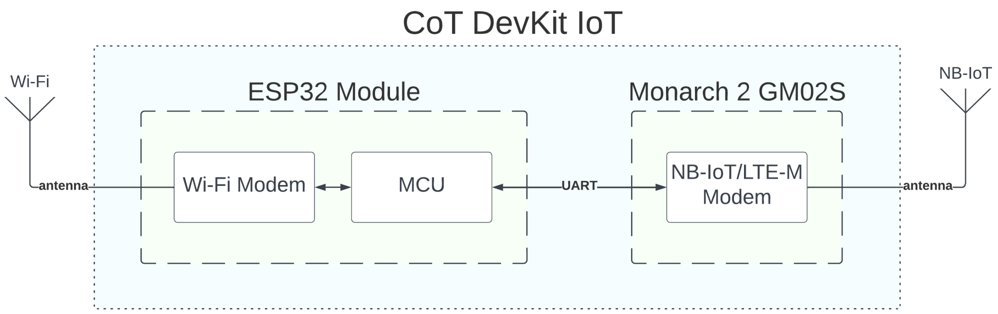
CoT DevKit IoT: Internet of Things hardware with Wi-Fi and narrowband IoT capabilities specialised for use in the Norwegian educational system

Electrical Engineering, Electronics and sensor systems

Main supervisor: Arne Midjo

August 2023

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems



NTNU | Norwegian University of Science and Technology

Company of Things

Project title:

CoT DevKit IoT: Internet of Things hardware with Wi-Fi and narrowband IoT capabilities specialised for use in the Norwegian educational system

Prosjekt tittel:

CoT DevKit IoT: Tingenes internett maskinvare med Wi-Fi og narrowband IoT funksjonaliteter spesialisert for bruk i det norske utdanningssystemet

Author:	**Submitted:**	**Pages/Appendices/Attachments:**	**Grade:**
Annik Riise	20.08.2023	84/14/15	[X] open [ ] closed

Study programme: Bachelor in Electrical Engineering

Field of study: Electronics and sensor systems

Supervisors:

Arne Midjo (Main supervisor), Department of Electronic Systems NTNU

Lars Lundheim (Co-supervisor), Department of Electronic Systems NTNU

Nils Kristian Rossing (Co-supervisor), Skoleaboratoriet, Department of Physics NTNU

Client: Company of Things AS

Contact: Aasmund G. Nørsett, aasmund@cot.as

Executive summary:

Through this bachelor thesis, a proof-of-concept of the "CoT DevKit IoT" was built. CoT DevKit IoT is a development board with Wi-Fi and NB-IoT capabilities suited for entry-level projects. The proof-of-concept's main components are the microcontroller ESP32-WROOM-32E with built Wi-Fi and the NB-IoT modem Monarch 2 GMS02. The components were compatible, and the demonstration setup showcased the network functionalities, and low-power modes and proved to be cost-effective. Main improvement areas include more complex hardware flow control, better power consumption measurements, and further NB-IoT data transmission testing before beginning the PCB design.

Kortfattet sammendrag:

Gjennom denne bacheloroppgaven ble det bygget et konseptbevis av "CoT DevKit IoT". CoT DevKit IoT er et utviklingsbrett med Wi-Fi- og NB-IoT funksjonaliteter som passer godt for nybegynnerprosjekter. Konseptbevisets hovedkomponenter er mikrokontrolleren ESP32-WROOM-32E med innebygd Wi-Fi og NB-IoT-modemet Monarch 2 GMS02. Komponentene var kompatible, og demonstrasjonsoppsettet viste nettverksfunksjonalitetene, laveffekts modusene og viste seg å være kostnadseffektivt. De viktigste forbedringsområdene inkluderer mer kompleks maskinvareflytkontroll, bedre målinger av strømforbruk og ytterligere testing av NB-IoT-dataoverføring før PCB-designet starter.

Keywords:	**Stikkord:**
Internet of Things (IoT), Wi-Fi, Narrowband IoT (NB-IoT), development board, proof-of-concept, Norwegian education, ESP32, Monarch 2 GM02S	Tingenes internett (IoT), Wi-Fi, Narrowband IoT (NB-IoT), utviklingskort, konseptbevis, norsk utdanning, ESP32, Monarch 2 GM02S

Table 1: Thesis information

Abstract

This report documents the creation of a proof-of-concept for the "CoT DevKit IoT", a versatile Internet of Things (IoT) development board with Wi-Fi and narrowband IoT (NB-IoT) capabilities. It is specialised for entry-level projects in the Norwegian educational system. The project is officially launched with this bachelor's thesis, and its continued development will be made possible by its conclusion.

IoT is a significant part of the present and future landscape of electronics and should, therefore, be incorporated into teaching in an engaging way. However, educational institutions frequently lack the funds, equipment, and time necessary to do this. As of March 2023, there are no single-board devices that meet the desire for both short- and long-range wireless communication (Wi-Fi and NB-IoT), are beginner-friendly for upper secondary and university students, have low power consumption, and are cost-effective to match limited school budgets.

The approach to fabricating the proof-of-concept involved selecting technology and system requirements based on feedback from educators at a conference and significant market research, as well as iteratively implementing the various elements and testing them individually before integrating them together. This resulted in a working product with the stated network functionalities, low-power modes, and component compatibility. The main components of the product are the microcontroller module ESP32-WROOM-32E with built-in Wi-Fi and the NB-IoT modem module Monarch 2 GMS02.

The primary areas for future work include more complex hardware flow control and employing proper power consumption measurements. Furthermore, more comprehensive NB-IoT data transmission testing is needed, as limitations from the original plan occurred. During the semester, a potential competition emerged – a development board that employs similar components. Despite this fact, it does not cater to the same intended market.

Throughout the course of this project, a successful proof-of-concept of the CoT DevKit IoT development board was built.

Sammendrag

Denne rapporten dokumenterer utviklingen av et konseptbevis for "CoT DevKit IoT", et allsidig utviklingsbrett for Tingenes Internett (IoT) med Wi-Fi- og narrowband IoT (NB-IoT) funksjonaliteter. Den er spesialisert for nybegynnerprosjekter innen det norske utdanningssystemet. Prosjektet blir offisielt lansert med denne bacheloroppgaven, og dets videre utvikling vil bli muliggjort gjennom oppgavens konklusjon.

IoT er en betydelig del av dagens og fremtidens elektronikklandskap, og bør derfor integreres på en engasjerende måte i undervisningen. Imidlertid mangler utdanningsinstitusjoner ofte midler, utstyr og tid til å gjøre dette. Per mars 2023 finnes det ingen enkeltbrettsenheter som tilfredsstiller behovet for både kort- og langdistanse trådløs kommunikasjon (Wi-Fi og NB-IoT), er brukervennlige for videregående skole og universitetsstudenter, har lavt strømforbruk og er kostnadseffektive nok til å passe innenfor begrensede skolebudsjetter.

Tilnærmingen til å lage konseptbeviset innebar å velge teknologi og systemkrav basert på tilbakemeldinger fra lærere på en konferanse, samt omfattende markedsundersøkelser. Videre ble de ulike elementene iterativt implementert og testet individuelt før de ble integrert sammen. Dette resulterte i et fungerende produkt med de angitte nettverksfunksjonene, laveffekts moduser og komponentkompatibilitet. Hovedkomponentene til produktet er mikrokontroller modulen ESP32-WROOM-32E med integrert Wi-Fi og NB-IoT modem modulen Monarch 2 GMS02.

De viktigste områdene for fremtidig arbeid inkluderer mer kompleks maskinvare kontroll flyt og ordentlige strømforbruk målinger. Videre er det behov for grundigere testing av NB-IoT-dataoverføring, da det oppstod begrensninger i forhold til den opprinnelige planen. I løpet av semesteret dukket det opp en potensiell konkurrent – et utviklingskort som bruker lignende komponenter. Til tross for dette, retter den seg ikke mot det samme tiltenkte markedet.

I løpet av dette prosjektet ble det bygget et vellykket konseptbevis av CoT DevKit IoT utviklingsbrettet.

Acknowledgements

This project was made possible with experience and knowledge gained from the bachelor program in electrical engineering with specialisation in electronics and sensor systems at NTNU. During the project development, the following people provided significant assistance.

I would like to thank my main supervisor, Arne Midjo, for providing great guidance along the way and especially for your expertise in engineering project development. Furthermore, thank you, Arne, for your support when I fell ill. Thank you to co-supervisor Lars Lundheim for quality feedback, particularly your knowledge of educating engineers. Nils Kristian Rossing, you have been a great co-supervisor. Having you as a sparring partner whenever I was stuck was invaluable, especially since you have a grasp of working with narrowband IoT. I would like to thank Ingulf Helland for assisting in figuring out how proper power consumption measurements could be done with the sensitive components.

I would like to thank Adrian Heyerdahl, whom I was initially working with before splitting the bachelor group, for being a great partner for collaboration, for everything administrative you did in the beginning, for pushing myself and the project forward, and for being such a good leader for Company of Things. Furthermore, I would like to thank everyone else at Company of Things, including Aasmund G. Nørsett, Tobias Domaas, Andreas Ødegård, Trym B. Aasheim, Sebastian Kalland, Erlend Storsve, and Thomas H. Tangen for creating such an exceptional company that was able to provide this project.

Annik Riise

Annik Riise
Trondheim, 20.08.2023

List of figures

Abbreviations and terminology

Add-on component - Hardware that enhances the capabilities or performance of other hardware

AT commands - "Attention" commands used to send intructions used to control a modem

DCE - Data Communications Equipment

Development board - A microcontroller integrated on a PCB for easier usage and access

DTE - Data Terminal Equipment

DTE-DCE connection - Interface where DTE is the master and DCE the subordinate

eDRX - extended Discontinuous Reception

ESP32 - Used as name for microcontroller module ESP32-ESP32-WROOM-32E and ESP32 DevKitC V4

Hardware flow control - Method in which a serial device controls data transmission to itself

Integration testing - Checking individual components of software to find problems or to verify that they work together

IoT - Internet of Things

IoT-platform - A website or software application that supports interacting with microcontrollers/development boards over the internet

IP address - Internet Protocol address
Iterative design - Cyclic process of prototyping, testing, analysing, and refining a product or process
Low-power modes - Reduce amount of power consumptions at different levels (ex. deep sleep)

LTE - Long-term evolution cellular network

LTE-M - Long-term evolution machine type communication cellular network

MCU - Microcontroller unit

Microcontroller - Integrated circuit designed to govern a specific operation in an embedded system (can be seen as a small computer)

Modem - Hardware that converts data from a digital format into a format suitable for an analog transmission (often wireless)

Module - Functional unit or component that is part of a larger system

Monarch 2 - Used as name for Monarch 2 GM02S Module and Monarch 2 GM02S NEKTAR Evaluation Kit

NB-IoT - Narrowband Internet of Things protocol using cellular low-power wide area network technology

PCB - Printed Circuit Board

Proof-of-concept - Demonstration of a product

PSM - Power Saving Mode
SoC - System on chip

UART - Universal Asynchronous Receiver/Transmitter (hardware communication protocol)

Unit testing - Small testable parts of an application

USB - Universal Serial Bus

Wi-Fi - Wireless networking technology that uses radio waves to provide wireless high-speed Internet access

Table of contents

1 Introduction

This chapter gives an introduction to the project and its background. The executive summary of the project can be read in Table 1, and the thesis poster is located in Appendix B.15.

1.1 Background and motivation

Internet of Things (IoT) is a central subject within the present and future of electronics. Here physical objects are connected together to facilitate data change to facilitate more efficient and sustainable systems. Given its vital role, it is important to include it in education.

This custom bachelor project was created in collaboration with Company of Things AS (CoT). CoT is a startup that produces different products, learning materials, and lectures focused on electronics and IoT for educational purposes. CoT aims at making it easier to integrate these themes into subjects by assisting the Norwegian school system, which lacks resources, time, and tools. They want to encourage a more practical learning approach and student engagement.

CoT has not produced any hardware devices yet and therefore wants to begin the design of a development board. This leads to the problem statement/project goal of this thesis shown below. The development board has been given the temporary name "CoT DevKit IoT". It will eventually be a complete circuit board that includes various components, interfaces, and firmware necessary to easily create electronics projects. CoT would like this board to be compatible with the IoT platform "Deploii" that they are developing.

Problem statement/Project goal:
"Create a proof-of-concept of an adaptable Internet of Things (IoT) development board, customised to cater to the specific demands and needs of the Norwegian education system."

1.2 Requirements and objective

The requirements and objectives are based on the problem statement/project provided above. The objective is to create a proof-of-concept of the CoT DevKit IoT that align with the functionality requirements given by Company of Things. To meet these functionality requirements, specific technologies were selected, and system requirements were formulated through this project. The following bullet list describes these.

System requirements for proof-of-concept of CoT DevKit IoT:

- Short-range wireless communication for indoor use: Wi-Fi.
- Long-range wireless communication for outdoor use: Narrowband IoT.

- Entry-level microcontroller programming language: Arduino C.
- Possibility for low power modes.
- Strive for low cost.

1.3 Report structure

The project report is divided into eight main chapters.

Chapter 1, "Introduction", gives an introduction to the project and its background.

Chapter 2, "Theoretical framework", explains some necessary theoretical themes that are imperative to understand the project but are not expected as a prior requisite for the reader.

Chapter 3, "Method", describes project management, approach and choice of technologies, system requirements, design, and test methodologies.

Chapter 4, "System design", first gives an overview of the system architecture, then elaborates on the hardware and software design and functionality, and lastly discusses reasonings behind some of the design choices.

Chapter 5, "Implementation and testing", describes the process of implementing the system and testing it.

Chapter 6, "Results", showcases the results from testing the final system setup, as described in the previous chapter.

Chapter 7, "Discussion", examines the project results and discusses other aspects related to it, such as how it could be implemented in subjects and future work.

Chapter 8, "Conclusion", restates the problem statements/project goal and addresses which requirements have been accomplished, challenges, project process, and what to prioritise as future work.

1.4 Who should read this report

This report is intended to be read by individuals with experience and knowledge within electrical engineering, equivalent to a minimum of three years of university education. It explains the theory that not everyone within that field is familiar with but subsequently does not dive deep into topics that are expected to know.

2 Theoretical framework

This Chapter elaborates on the central technologies and theoretical background needed to understand this project report.

2.1 Internet of Things

The phrase "Internet of Things" (IoT) encompasses the idea of establishing interconnected networks composed of different "things" to enhance efficiency [1]. Figure 1 illustrates this. "Things" refer to different physical entities equipped with sensors, software, and connectivity capabilities for data exchange and acquisition between them while keeping low-power consumption. IoT is a central part of the fourth industrial revolution known as "Industry 4.0" [2]. Implementations of IoT systems are vast and include industrial machinery, vehicles, household and human wearable appliances, and entire smart cities. Electronics engineers play a crucial role in this field, as they can design and develop both hardware and software for IoT devices [3]. Given IoT's significance for the future, education should incorporate IoT.

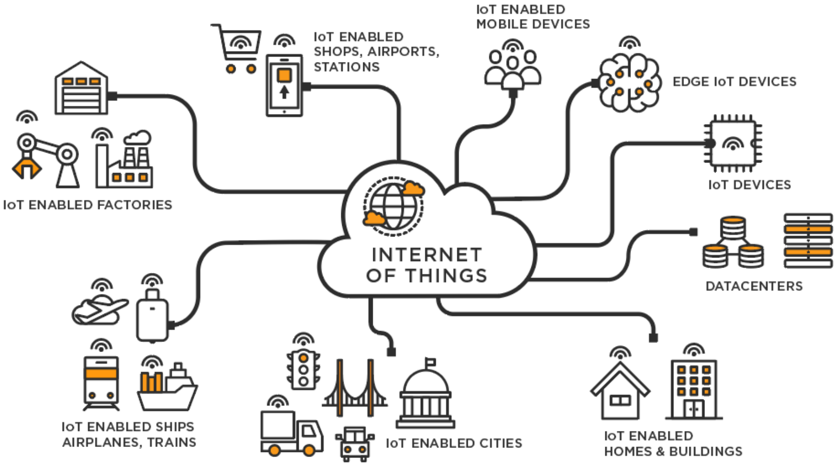

IoT ENABLED SHOPS, AIRPORTS, STATIONS
IoT ENABLED MOBILE DEVICES
EDGE IoT DEVICES
IoT DEVICES
IoT ENABLED FACTORIES
INTERNET OF THINGS
DATACENTERS
IoT ENABLED SHIPS AIRPLANES, TRAINS
IoT ENABLED CITIES
IoT ENABLED HOMES & BUILDINGS

Figure 1: Internet of Things illustration [4]

2.2 Microcontrollers

Microcontrollers serve as a central technical element within the realm of IoT. These compact integrated circuits are specifically designed to control specific operations within embedded systems [5]. They are commonly referred to as an MCU (microcontroller unit). A typical microcontroller consists of a processor, memory, and input/output (I/O) peripherals, all housed on a single chip. Being able to utilise microcontrollers is essential for developing IoT systems. C and C++ are the most commonly used programming languages, but others can be employed [6].

2.3 Wi-Fi

Wi-Fi is a wireless network technology that uses radio waves to connect devices to the internet [7]. Operating based on the IEEE 802.11 standard, which serves as the foundation for wireless local area networking. It commonly uses the frequency bands 2.4 GHz and 5 GHz.

The two main components within a Wi-Fi network are access points (APs) and clients [7]. Access points host the local network where clients can connect. Examples of client devices are laptops and smartphones, while an AP can be a Wi-Fi router in a home. To ensure data security, the transmission of data between the client and the access point is encrypted. Figure 2 shows a common connection request and approval between an AP and a client.

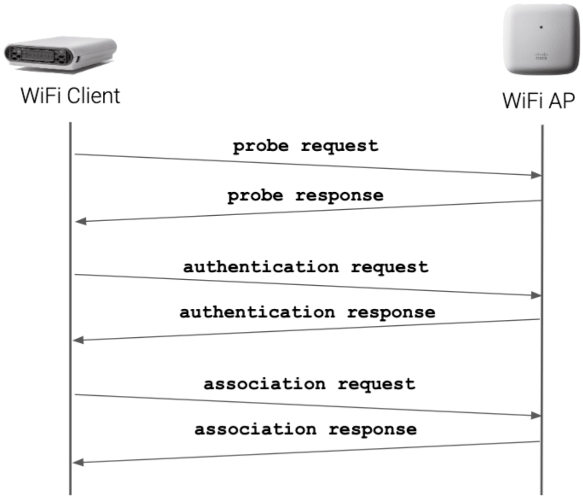


Figure 2: Wi-Fi connection request and response [7]

2.4 Narrowband Internet of Things (NB-IoT)

Narrowband IoT (NB-IoT) is a wireless communication protocol that has been specifically devised to cater to the requirements of the Internet of Things (IoT) domain. It leverages the utilisation of low-power wide-area network (LPWAN) technology, which enables the transmission of limited data volumes over a constrained bandwidth, thereby conserving substantial amounts of power [8]. NB-IoT operates within cellular networks and can be classified as a subset of long-term evolution (LTE) wireless broadband. To put it simply, LTE corresponds to the 4G network, while LTE-M refers to an optimised variant of 4G [9]. Notably, 4G pertains to the fourth generation of broadband cellular network technology. Furthermore, it is worth mentioning that NB-IoT is also anticipated to be made available within 5G networks [10]. Alternative terminologies and specifications associated with NB-IoT encompass LTE Cat-NB1, and LTE Cat-NB2 [8].

2.4.1 AT commands

AT commands (short for "Attention") are used to control modems [11]. These are often entered in a terminal window on a PC, where the device is connected to a serial port. If "AT" is typed and it returns "OK", this indicates that they are connected properly. AT commands have a few classifications and syntaxes.

AT command type:	Syntax:	Description:
Test	AT+command name=?	Check if command is supported by the modem and possible parameter values
Read	AT+command name?	Check current parameter values and settings
Write	ATcommand name	Basic command, no "+" prefix
	ATcommand name=xx	Basic command with sub parameter
	AT+command name	Extended command, prefix with "+"
	AT+command name= , , xx	Some sub parameters may be omitted
	AT^command name	Some private commands use other characters instead of "+", e.g. "AT^RESET"

Table 2: AT command classifications and syntaxes [11]

2.5 Low-power modes and techniques

When designing IoT systems and devices, a crucial aspect includes minimising power consumption. Devices often have different techniques to enter different power-saving modes. Distinguishing techniques and modes are important. Modes refer to distinct power states that offer different available functionalities, whereas techniques entail methods for transitioning into and out of these states. Employing techniques and modes instead of merely shutting devices off ensures that different configurations are not lost and keeps some services.

2.5.1 Low-power modes

Unnecessary power consumption is not ideal and sustainable in many projects. Having a balance between power-saving and required functionality operations becomes a central part of the design. Devices often have their own specific power-saving modes where they deactivate some elements and can transition between them as needed [12]. For instance, if some functionalities are only required at certain times, it is nonessential to keep all functionalities active continuously. Terms such as "idle" and "sleep" are frequently employed to describe different low-power states. "Idle" often refers to devices not being in use and goes, for example, into "deep sleep" when it has been in idle for an extended period of time, where it shuts down even more elements [13]. Detailed information regarding specific modes relevant to the components utilised in this project can be found in their corresponding descriptions in Chapter 4.2.4 and 4.2.5.

2.5.2 Low-power techniques on cellular network

Employing power-saving techniques, as opposed to merely shutting down the modem, ensures that the device, for example, deactivates radio functionalities while retaining connectivity to the network without necessitating the power-intensive attachment procedure for reconnecting. There are two main power-saving techniques when considering cellular networks (including NB-IoT): Power Saving Mode (PSM) and extended Discontinuous Reception (eDRX) [14]. In essence, PSM and eDRX differ in terms of the depth and duration of sleep. PSM allows for significantly longer sleep duration, enabling the device to enter a deeper sleep state that consumes less power. However, this comes at the cost of a longer wake-up period. In the case of PSM, "sleep" refers to the state where both the receiver (RX) and transmitter (TX) of the radio are turned off while the device remains registered with the network. On the other hand, eDRX involves turning off the TX and intermittently activating the RX to listen for paging messages that inform the device that there is data pending. Enabling eDRX to listen for incoming data more often consumes more power. PSM and eDRX can also be combined. When going to sleep, the connection to the network is turned off, referring to RRC idle (Radio Resource Control).

Power Saving Mode (PSM):
PSM is an optimal choice when devices can operate independently for prolonged durations without network interaction [15]. It is explicitly designed for IoT devices. Both the network and device must support this functionality to enable PSM. PSM refers to a state wherein the device minimises its power consumption by entering a sleep mode while maintaining connectivity configurations. During PSM, the device ceases to monitor paging, rendering it unreachable.

The device periodically wakes up and transmits a Tracking Area Update (TAU) to the network. TAU is a signaling procedure to inform the network about its location to optimise resource allocation and ensure efficient communication. Here, two timers are involved: Periodic TAU timer (T3412) and Active Timer (T3324). The Periodic TAU timer determines the duration between these device awakenings. The Active Timer (T3324) defines the time at which the device is reachable. The difference between these two is the duration for which a device remains in Power Saving Mode. This is visualised in Figure 3, where the x-axis is power consumption and user equipment (UE) refers to the device. However, the device can also be triggered to be woken up by other events, such as sensor activation or another external request, for example, a pin toggle.

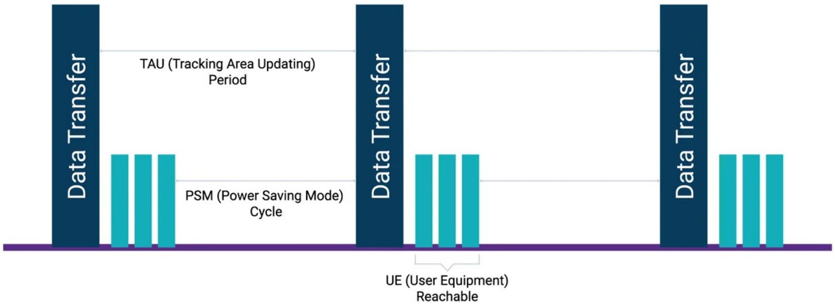

Data Transfer
TAU (Tracking Area Updating) Period
PSM (Power Saving Mode) Cycle
Data Transfer
Data Transfer
UE (User Equipment) Reachable

Figure 3: PSM cycle [15]

extended Discontinuous Reception (eDRX):
eDRX, an extension of DRX and closely related to PSM, offers enhanced flexibility in defining the cycles [15]. eDRX monitors the network more frequently than PSM, although at the expense of reduced energy savings. Whereas PSM cycles can span over several days, eDRX cycles have a maximum duration of a few hours but typically operate in seconds and minutes.

The Paging Cycle Length (PCL) parameter determines the interval at which network monitoring occurs, while the Paging Transmission Window (PTW) parameter specifies the duration of this monitoring. The device and the network engage in a negotiation process to establish a specific sleep cycle duration. In Figure 4, the eDRX cycle is depicted, with the x-axis representing power consumption. The light blue column is PTW, and that column plus the eDRX cycle is the PCL.

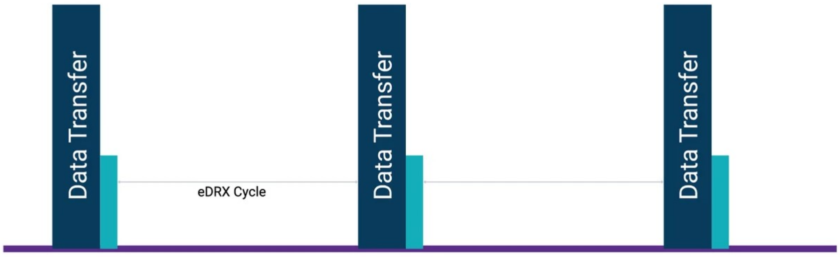


Figure 4: eDRX cycle [15]

2.6 Pedagogical principles when creating a development board

During the fabrication process of the development board with its intended use in education, it is important to keep pedagogical principles in mind. This is also essential when creating different assignments.

Project-based and hands-on learning approaches stimulate active and experiential learning, contrasting with passive methods, by offering a practical dimension that enhances student engagement [16].

Contextual learning arises when students acquire proficiency in utilising the development board while simultaneously gaining knowledge about various real-life applications [17]. Within the context of IoT, this might be smart homes or environmental monitoring.

Assignments that are problem-centered [18] and conducted in teams [19] necessitate the utilisation of reflective, analytical, and problem-solving skills among students while concurrently fostering collaboration and enhancing communication abilities.

Providing board flexibility and adaptability allows students to explore and properly customise their own projects [20]. This is an important aspect to consider during its development. This also translates to that it should not be too complected to begin using.

By implementing these principles in the fabrication of the board and assignments, students not only grasp the curriculum but also nurture critical thinking, problem-solving skills, creativity, and collaboration.

3 Method

This chapter provides a comprehensive overview of the methods utilised to implement the project structure and design and why these were chosen. It includes the project's organisational aspects, system, requirements, design methodology, the reasoning behind the selection of the primary components, and testing methods employed. After this chapter, the reader should have an informational overview to understand the further and actual implementation of the project.

3.1 Project planning and management

Project planning and management methods were established with the pre-project report. However, in the beginning, the project group consisted of two members. Due to one of them falling ill, the decision to split up the project was made so that the other could submit it on time. From the beginning, the project consisted of two pretty separate parts, therefore presented the division no great challenge. One part of the project was developing the IoT platform Deploii, while the other part was the proof-of-concept of an IoT development board. Evidently, this project report encompasses the latter. Attending the Technology and Research Education Conference was an overlapping task and was therefore kept in both. It is worth mentioning that the project developer, Annik Riise, is the electrical engineering student carrying out this bachelor project on behalf of Company of Things but is also the company's COO. All documents related to the project were added to the Microsoft Teams for this bachelor thesis, available for supervisors. These are attached to the submission. See B.1.

3.1.1 Goals

The main project goal, presented in Chapter 1.1, states: "Create a proof-of-concept of an adaptable Internet of Things (IoT) development board, customised to cater to the specific demands and needs of the Norwegian education system". Based on this and other wishes from Company of Things, a small list has been created, as shown below. The work packages in the next Subchapter, "3.1.2 Quality and progress ensures with work packages", can also be seen as goals.

- Hardware for proof-of-concept of CoT DevKit IoT
- Software/code for proof-of-concept of CoT DevKit IoT
- Able to work with the IoT-platform Deploii
- Final project report
- Final presentation and poster

3.1.2 Quality and progress ensured with work packages and Gantt-chart

To keep track of time and ensure progress and quality, the project was divided into smaller subtasks called work packages. The work packages are based on the system requirements, goals stated, and wishes of Company of Things. The choice of type of main technological components was done prior to the pre-project report. Therefore are specifications included in the work packages. Reasoning for them can be read about in Chapter "3.4 Choice of main technological components". The bullet list below showcases the different work packages. The "H" stands for "Hardware", "OF" stands for "Outreach and Feedback", and "R" stands for "Presenting results".

- H1: Decide on microcontroller and NB-IoT modem
- H2: Test Wi-Fi and narrowband data transfer and communication
- H3: Test data transfer and communication with the IoT-platform (Deploii)
- H4: Possible low-power mode
- OF1: Technology and Research Education Conference
- P1: Final report
- P2: Final presentation and poster

The Gantt chart in Appendix A.1 visualises the work packages, the planned duration, and actual progress. Most tasks used approximately the expected time period. Planning for OF1 took longer than anticipated, causing a delay in H2 startup. H3 was not done, as the IoT platform Deploii was not ready throughout the project period. The project developer was sick from week 19 through week 21 when no work was done on the project. After this, the progress was slower due to not being completely well and working at half capacity. This resulted in a total delay of approximately two months.

3.1.3 Quality and progress ensured with supervisors

Academic support and feedback from supervisors allowed for constant improvement and progress, keeping the project on track and ensuring quality. Having these sparring partners also contributed to the best solutions. In the first half of the semester, meetings were held every other week, and in the second half of the semester, every week. One supervisor, Even J. Christiansen, a staff engineer at the Department of Electronic System NTNU, dropped out due to now having enough time. This somewhat negatively affected the project, as he knows quite a bit about microcontrollers. However, questions were still directed towards him. Additionally, Ingulf Helland, a senior engineer in the same department, provided assistance with hardware questions. The other supervisors provided great feedback and assistance with their knowledge of wireless communication, Arduino programming, electronics, pedagogical methodology, and general project management. Below is a list of the supervisors.

- Arne M. Midjo (Main supervisor), Department of Electronic Systems NTNU

- Lars M. Lundheim (Co-supervisor), Department of Electronic Systems NTNU
- Nils Kristian Rossing (Co-supervisor), Skoleaboratoriet, Department of Physics NTNU

3.1.4 Risk assessment

The project has no HSE (Health, Safety and Environment) risks since the work was done on computers and with low-power circuitry. Interpreting "risk" as a broader term, the main risk was keeping to the laid-out plan. This is due to the project being created from passion, and the project developer might get stuck on certain details.

3.1.5 Time and cost

Cost consists of equipment and man-hours. The final expenses and when they occurred, is visualised in the S-curve chart in Appendix A.2.

The price of equipment was estimated to be 3800 NOK, while the actual material expenses ended up on 3072 NOK (excluding transport), as depicted in the list below.

- 2x Development board with microcontroller that has integrated Wi-Fi, 210 NOK.
- 2x Evaluation kit with NB-IoT module, 2162 NOK.
- 2x SIM card with NB-IoT subscription, 50 NOK each month (4x50 NOK).
- Other (breadboard, cables, battery), 500 NOK.

Starting salary for newly graduated electrical engineers in Norway is typically 550 000 NOK [21]. This includes 48 work weeks a year, each week consisting of 5 days, where one day is 7,5 hours. This results in 1800 hours a year with an hourly salary of 550 000/1800 = circa 305 NOK. Due to other commitments, the project developer worked 3 days a week from January through March, and after this, 5 days a week. The conference was in week 11 in March, therefore all 5 days went to this. The project developer has exams in May and will therefore miss 12 workdays. This was the original plan, was that the project lasted from 9th of January until 4th of June, leading to a predicted $(11 weeks \cdot 3 days \cdot 7,5 hours) + (5 days \cdot 7,5 hours) + (10 weeks \cdot 5 days \cdot 7,5 hours) - (12 days \cdot 7,5 hours) = 570$ hours => 173 850 NOK. The expected cost, combining material and man-hour expenses, was 177 650 NOK.

As stated, the project progression went according to plan from January through the first week of May. After that, there was 3 weeks without work, and following this worked at half capacity. Additionally, week 26 went to another job. The project was done in week 31. This results in total cos of man-hours to be $(11 weeks \cdot 3 days \cdot 7,5 hours) + (5 days \cdot 7,5 hours) + (5 weeks \cdot 5 days \cdot 7,5 hours) - (12 days \cdot 7,5 hours) + (9 weeks \cdot 3 days \cdot 7,5 hours) =$ 585 hours => 178 425 NOK.

The actual cost of the project, with 3072 NOK for equipment and 178 425 NOK for man-hours (585 hours), ended up at 181 497 NOK.

3.2 System requirements

The system requirements of the project are primarily derived from the specific needs of the Norwegian education system and the preferences of Company of Things. This subchapter provides a comprehensive explanation as to why these chosen technologies constitute system requirements for the project, and they are presented in the concluding section of this subchapter. Therefore, the "Functionality requirements" refer to the specifications and functionalities requested by Company of Things. While the "System requirements" delve deeper into the specific technologies and components are chosen by the project developer to fulfill the functionality requirements.

First, it is important to reflect on the reasons behind Company of Things' specific requirements for CoT DevKit IoT. The scope of Internet of Things (IoT) extends to both indoor and outdoor settings, including remote areas, necessitating the utilisation of wireless data transfer technologies that cater to both short and long-range communication. Consequently, the development board should be purposefully designed to accommodate entry-level projects, thereby influencing the selection of an appropriate programming language and level of complexity. In the context of IoT, the capability to conserve energy through low-power options is of paramount importance to ensure product sustainability and longevity. Given that the projects undertaken by students are not expected to be excessively large in scale, high bandwidth requirements for data transfer are not imperative. These functional requirements are succinctly summarised in the following list.

Functionality requirements for proof-of-concept of CoT DevKit IoT:

- Short-range wireless communication for indoor use.
- Long-range wireless communication for outdoor use.
- Entry-level microcontroller programming language.
- Strive for low-power consumption.
- Strive for low cost.
- No need for highest data transmission.

With the list above in mind, there are a few possible wireless communication and programming language option to be considered.

Possible technologies:

- Wireless communication technology:
  - Short-range: Bluetooth or Wi-Fi
  - Long-range: Cellular 4G/5G, LoRaWAN or Cellular narrowband IoT
- Programming language:
  - C/C++, Arduino C or Python/microPython

Below are the concluding system requirements. The rest of this subchapter explains why these were chosen.

System requirements for proof-of-concept:

- Short-range wireless communication for indoor use: Wi-Fi.
- Long-range wireless communication for outdoor use: Narrowband IoT.
- Entry-level microcontroller programming language: Arduino C.
- Possibility for low power modes.
- Strive for low cost.

3.2.1 Technology and Research Education Conference

The Technology and Research Education Conference held in March at Arendal Upper Secondary School provided valuable insights into the preferences and inclinations of teachers regarding the integration of Internet of Things projects into their lectures. This annual conference is organised by the Schoollaboratory at NTNU and Tekna and is specifically targeted at upper secondary school teachers instructing the subject "Technology and Research" (Teknologi og Forskningslære) from all over the country. As representatives of Company of Things, one of my colleagues and I participated in the conference and delivered presentations, and gained feedback on products being developed. Our data acquisition methods included direct conversations with the teachers as well as a survey.

To illustrate the outcomes of inquiries related to this thesis, three questions from the survey have been selected. The graphs are presented in Figures 5, 6, and 7 below. Larger versions of the images can be found in Appendix A.3. The questions are, translated from Norwegian, as follows: "Which wireless technologies are you interested in utilising for educational purposes, ideally?", "Which development boards/microcontrollers do you prefer to use in your teaching?" and "Which programming languages do you prefer to use in your teaching?". By engaging in direct conversations with the teachers, we gained a deeper understanding of the reasoning behind their responses. It is important to acknowledge that the sample pool for the survey was relatively small, with 10-12 responses, and therefore the findings should be interpreted with caution. However, this is a good starting point.

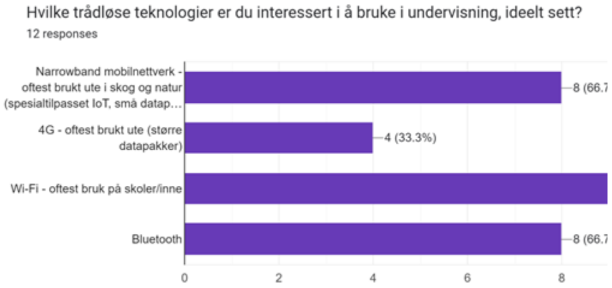


Figure 5: Survey question 1

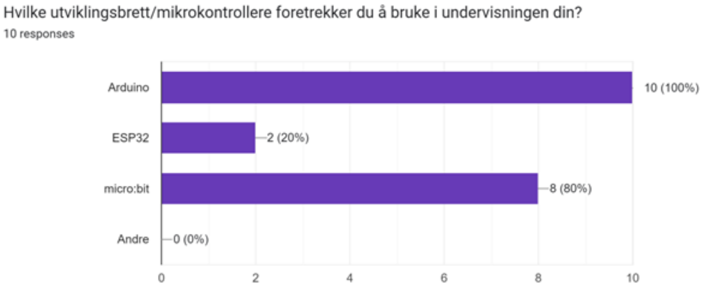


Figure 6: Survey question 2

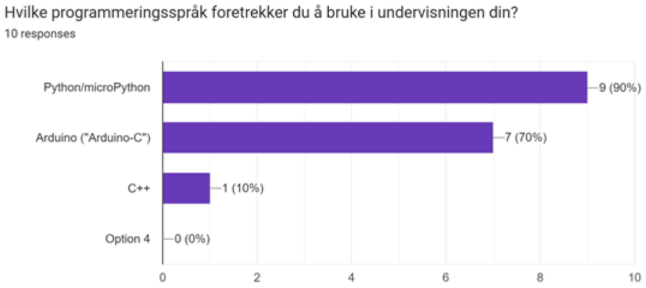


Figure 7: Survey question 3

There seems to be a correspondence between the types of programming languages and wireless communication technologies and the microcontroller boards employed. For instance, Arduino and ESP32 boards utilise Arduino C, while Micro:bit boards are compatible with Python/microPython. Micro:bit predominantly employs Bluetooth, whereas ESP32 offers both Wi-Fi and Bluetooth capabilities. On the other hand, Arduino Uno, the most commonly used Arduino board, lacks access to either of these connectivity options. Nevertheless, a few teachers informed us that they had procured additional components to enable Wi-Fi connectivity for their Arduino boards. A lot of the schools already have Micro:bit and Arduino broads. ESP32 is gaining popularity in higher education as it is perceived as more cost-effective, secure, and versatile than Arduino boards, thanks to its Wi-Fi and Bluetooth capabilities. This trend might be transferred to upper secondary education. A noteworthy finding from the survey is that none of the teachers currently employ narrowband, although they express a keen interest in incorporating it into their teaching.

3.2.2 Wireless communication and programming language

Short-range wireless communication:
The technologies being examined are Bluetooth and Wi-Fi. Bluetooth operates on the 2.4 GHz radio frequency band and enables device connectivity within a specified range

[22]. Wi-Fi allows devices to connect wirelessly to the internet and facilitates interconnection among devices operating in both the 2.4 GHz and 5 GHz frequency bands [7]. Data obtained from the survey at the Technology and Research conference indicates that teachers express a nearly equal desire to employ these resources, see Figure 6.

Compared to Bluetooth, Wi-Fi has a wider coverage range, encompassing distances ranging from tens to hundreds of meters depending on the specific equipment and environment [7]. In contrast, Bluetooth has a shorter range, with Bluetooth 4.2 reaching a maximum of 60 meters and Bluetooth 5.0 reaching 240 meters [22]. Notice that these distances can be significantly influenced by environmental factors and infrastructure, resulting in reduced range. Wi-Fi, compared to Bluetooth, has faster data transmission speeds. The latest version of Bluetooth has significantly improved, but it can still only transmit up to 2 megabits per second [22]. Whereas Wi-Fi can transmit up to a few gigabits per second [7]. An advantage of Bluetooth emerges from its narrower range and lower transmission speeds, as this results in lower power consumption. However, another disadvantage is that it is a less secure protocol [22].

In summary, Wi-Fi offers superior data transfer speeds, broader coverage, and enhanced security compared to Bluetooth. Nevertheless, the lower power consumption of Bluetooth and the absence of a requirement for very high data transmission may make it a viable option in certain scenarios. However, in this particular project, Wi-Fi is chosen based on a few factors. Wi-Fi provides indoor/short-range coverage that extends beyond a single classroom to encompass the entire school, which would be unreliable with Bluetooth. Additionally, secure data transfer is prioritised. This development board is specifically designed for seamless integration with the IoT platform, Deploii, developed by the Company of Things. The initial connection method being implemented with this platform is Wi-Fi. Furthermore, as stated, the survey suggests that teachers are indifferent about which of these two to use. Based on this, Wi-Fi is chosen as the short-range wireless communication technology. More about narrowband IoT can be read in 2.3 Wi-Fi, under the Theoretical Framework chapter.

Long-range wireless communication:
The long-range wireless communication technology options encompass 4G, 5G, LoRaWAN, and NB-IoT. 4G, or fourth-generation wireless, is a high-speed cellular network technology [23]. 5G, or fifth-generation wireless, represents the most recent iteration within this series, characterised by decreased latency, heightened capacity, and augmented data transfer swiftness compared to 4G [24]. LoRaWAN stands for Long Range Wide Area Network and is, as the name suggests, a long-range, low-power wide-area network (LPWAN) technology [25]. It is specifically designed for IoT applications and employs a proprietary spread spectrum modulation technique called LoRa. Meaning it has its own network. NB-IoT, short for Narrowband IoT, is also a low-power LPWAN technology created for IoT [8]. However, NB-IoT utilises the existing 4G infrastructure and is set to also work on the 5G network.

The key distinctions between 4G/5G, NB-IoT, and LoRaWAN lie in their cellular connectivity, data rates, power consumption, and range/coverage capabilities. 4G and 5G are high-speed technologies, while NB-IoT and LoRaWAN are LPWAN technologies optimised for low-power IoT devices. While both NB-IoT and 4G/5G utilise cellular networks and operate on licensed frequency bands, NB-IoT is a subset of 4G. In contrast, LoRaWAN operates on unlicensed frequency bands and does not rely on cellular infrastructure. 4G and 5G offer high data transfer rates suitable for applications that necessitate real-time streaming and high-bandwidth communication. NB-IoT and LoRaWAN provide lower data rates but have low power consumption and are ideal for connecting IoT devices over long distances. Considering the functional requirements of this project, the high data rates and power consumption of 4G/5G are not necessary. Additionally, teachers at the Technology and Research conference only showed minimal interest in using 4G (see Figure 5). Therefore, the choice comes down to LoRaWAN or NB-IoT.

LoRaWAN has a lower data rate and power consumption compared to NB-IoT. Although a high data rate is not crucial for this project, having really low limits students from creating more versatile products. Since the projects created by students may involve monitoring real-time data transmission, such as remote monitoring or control systems, NB-IoT is better suited. LoRaWAN boasts ultra-low power consumption, with devices often having a battery life of 15+ years, while NB-IoT offers a battery life of 10+ years [25]. However, these optimisations may not be necessary for student projects. Allowing students to develop more dynamic projects with slightly higher bandwidth is more valuable than achieving the lowest possible power consumption. One of the advantages of LoRaWAN is that users can operate their own network operator, thereby avoiding restrictions imposed by specific network operators. However, this also presents the main disadvantage of LoRaWAN: poor coverage. LoRaWAN adopts a star-of-stars network architecture, where end devices communicate with gateways that forward the data to a central network server. In Norway, there are not many gateways available, and schools may need to establish their own infrastructure. In comparison, NB-IoT utilises the existing 4G network, which offers extensive coverage in Norway. Maps of this can be seen in Figures 8 and 9. Note that the NB-IoT map is from Telenor's network, a Norwegian majority state-owned multinational telecommunications company and that the map consists of multiple images due to not being able to zoom out more. The coverage of NB-IoT is also superior indoors and in underground areas like cellars. Furthermore, LoRaWAN may face signal disturbances from building infrastructure. Teachers at the Technology and Research conference, even though they do not use NB-IoT today, seemed quite eager to include it in their lecturers (see Figure 5). Moreover, the market already offers a considerable number of LoRaWAN development boards. Taking all these factors into consideration, NB-IoT emerges as the most suitable choice. More about narrowband IoT can be read in 2.4 Narrowband Internet of Things (NB-IoT), under the Theoretical Framework chapter.

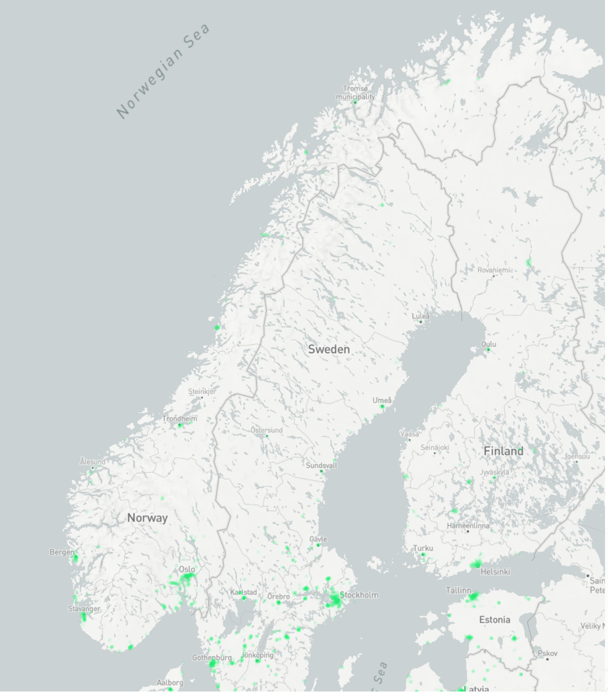


Figure 8: LoRaWAN coverage [26]

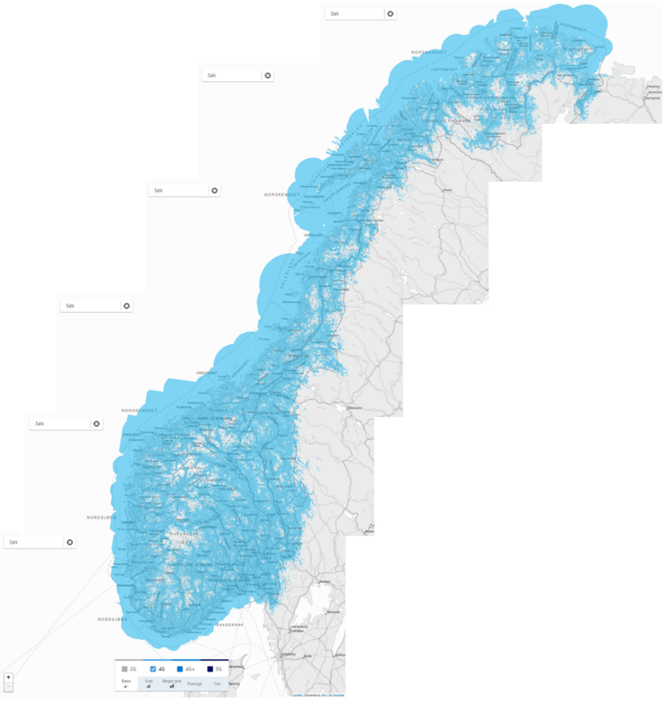


Figure 9: Telenor NB-IoT coverage [27]

Programming language and integrated development environment:
When embarking on the journey of learning how to use microcontrollers, selecting the most suitable programming language is a crucial decision. The options to consider are Python/microPython, C/C++, and Arduino C.

C and C++ are popular within microcontroller programming due to their ability to access hardware at a low level, deliver high performance, and have a wide range of existing libraries [28]. However, for beginners, the complexity of C and C++ can be overwhelming. In response to this, Arduino C was developed as a beginner-friendly language based on C++ [29]. Python is not traditionally a microcontroller programming language but it is becoming increasingly more attractive to use as such. Allowing individuals with Python experience to engage with microcontrollers without needing to learn a new language [30]. Notably, teachers at the Technology and Research conference exhibit a preference for Python and Arduino C, with Python slightly surpassing Arduino C (see Figure 7). Interestingly, when considering the preferred development board, Arduino only slightly outperforms Micro:bit (see Figure 6). This observation suggests that while Arduino boards are favored, Python is the most used language.

Nonetheless, C remains inherently more well-suited for microcontroller programming. For students delving deeper into the subject, transitioning from Arduino C to regular C/C++ is a more natural progression compared to transitioning from Python. Additionally, from a practical standpoint, the project developer for this thesis has more experience with Arduino C than Python.

In the case of utilising NB-IoT modems, which have been selected for this project, control is achieved through AT commands ("AT" stands for "attention"). These commands are entered into a terminal program on the computer. This aspect can also be slightly daunting for absolute beginners. Fortunately, Arduino offers its own integrated development environment known as Arduino IDE [31]. The IDE includes a tool called "Serial Monitor" which is a type of terminal but can, with minimal code, replicate the functionalities of standard terminal programs such as PuTTY and Tera Term [32]. This integration of code execution, manual command input, and display of responses within a single program fosters organisation and contributes to an improved workflow and user experience.

Considering all these factors, Arduino C with Arduino IDE emerges as the optimal choice for this project.

3.2.3 Concluding system requirements

Based on everything above, the functionality requirements can be specified into system requirements.

System requirements for proof-of-concept:

- Short-range wireless communication for indoor use: Wi-Fi.
- Long-range wireless communication for outdoor use: Narrowband IoT.
- Entry-level microcontroller programming language: Arduino C.
- Possibility for low-power modes.
- Strive for low cost.

3.3 Design methodology

The design methodology employed for creating this project was iterative design methodology. Iterative design is an approach that involves continuously improving a concept or product through a series of iterations [33]. See Figure 10 for a visual representation of this process. In the context of this project, the iterative design process was applied within each work functionality requirement. This approach involved working on individual subtasks, testing them, making adjustments, and repeating the process until satisfactory results were achieved. Once all the separate parts were completed, they were combined to form the final proof-of-concept with the demonstration setup.

It is important to note that the participants involved in this design cycle included the project developer, supervisors, employees at Company of Things, and potential teachers. However, it is worth mentioning that users or students were not directly involved in this particular design cycle.

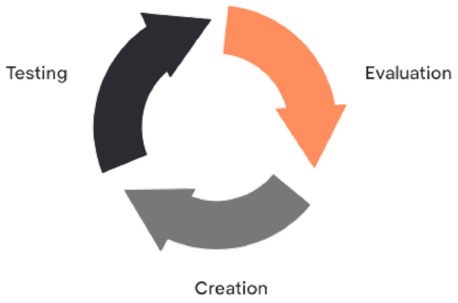


Figure 10: Iterativ design process [33]

3.4 Choice of main technological components

In order to develop a proof-of-concept aligned with the specified system requirements, it is essential to conduct a comprehensive analysis of the current market landscape and then choose the right components. The system's central components are a microcontroller that controls the entire system and narrowband and Wi-Fi communication modems. They need to be compatible with Arduino C and have low-power abilities. An additional aspect to consider is the objective of achieving a low-cost solution. Determining a reasonable price point and assessing the affordability within the school system present challenging questions to address. In this context, it becomes essential to select components that are both cost-effective and capable of meeting the specified requirements. The primary aim is to ensure maximum affordability for schools during the procurement process.

The market analysis was performed in March 2023, prior to the submission of the pre-project report. Consequently, it is important to acknowledge that there may have been subsequent changes in the market dynamics. The market analysis involved gathering information about existing development boards and compatible add-on components. Add-on components are boards that can be combined with other development boards to achieve the desired functionalities. Furthermore, moving on to the most important part of analysing existing microcontrollers and modems. When choosing which of these will be integrated in CoT DevKiT IoT.

3.4.1 Existing development boards and add-on components

Table 3 provides an overview of the available development boards that are pertinent to entry-level IoT projects, exhibiting certain functionalities that are aligned with the system requirements. Most of these do not include both narrowband IoT and Wi-Fi, meaning they would need to be combined with add-on components shown in Table 4 and 5, making them all together even more expensive.

The only available development board identified, possessing both narrowband IoT and Wi-Fi capabilities, which appears to be targeted towards entry-level applications and reasonably priced, is the LILYGO T-SIM7080G-S3 [34]. Consequently, it serves as a direct competitor to CoT DevKit IoT. Upon conducting online research, it is evident that the available documentation for this board is relatively limited. However, it should still be manageable as it utilises the ESP32-S3 microcontroller, which has sufficient resources and supporting documentation accessible [35]. It is worth noting that this particular board cannot be purchased through reliable distributors such as Mouser or Digi-Key, thereby raising concerns about the quality and reliability of the product. Nonetheless, Company of Things remains confident in its ability to compete with this competitor.

The applicable development boards with narrowband IoT were Arduino MKR NB 1500 [36], AVR-IoT Cellular Mini [37], and SparkFun Thing Plus [38]. However, both the SparkFun Thing Plus and Arduino MKR NB 1500 exhibit relatively high price points, respectively approximately 1000 NOK and 1500 NOK. This elevated cost can be attributed to them using expensive NB-IoT modems. On the other hand, the AVR-IoT Cellular Mini, despite being relatively expensive considering its sole narrowband IoT functionality, is priced lower compared to the two others. Furthermore, Microchip Technology provides comprehensive support resources for this particular board, enhancing its appeal and usability.

There are considerably more available boards with Wi-Fi. Therefore, the boards presented in Table 3 have been meticulously chosen as the most discerning options. The options are Arduino UNO WiFi REV2 [39], ESP32-DevKitC V4 [40] and Raspberry Pi Pico W [41]. Respectively priced at around 550 NOK, 110 NOK and 65 NOK, the Arduino UNO WiFi REV2 appears to be way overpriced. Both ESP32-DevKitC V4 and Raspberry Pi Pico W seem like quality boards. Something interesting to note is that the microcontroller on ESP32-DevKitC V4 has integrated Wi-Fi, while Raspberry Pi Pico W has separate microcontroller and Wi-Fi modem.

Existing Development Boards March 2023						
Dev Board:	**Manufacturer:**	**NB-IoT:**	**Wi-Fi:**	**Arduino C compatible:**	**Low-power:**	**Price in NOK:**
Arduino MKR NB 1500	Arduino	SARA-R410M-02B	–	✓	✓	934 [36]
Arduino UNO WiFi REV2	Arduino	–	NINA-W102	✓	✓	543 [39]
AVR-IoT Cellular Mini	Microchip Technology	Monarch 2 GM02S	–	✓	✓	746 [37]
ESP32-DevKitC V4	Espressif Systems	–	ESP32-WROOM-32E	✓	✓	108 [40]
LILYGO T-SIM7080G-S3	LILYGO	SIM7080G	ESP32-S3	✓	✓	320 [34]
Raspberry Pi Pico W	Raspberry Pi	–	Infineon CYW43439	✓	✓	65 [41]
SparkFun Thing Plus	SparkFun Electronics	nRF9160	–	✓	✓	1513 [38]

Table 3: Available Development Boards and some of their functionality

Moving on to the discussion of add-on boards, first looking at the narrowband options presented in Table 4. The LTE CAT M1/NB-IoT Shield for Arduino stands out as quite expensive, with a price close to 1000 NOK [42]. Additionally, Arduino boards themselves are already known for their relatively high costs. Similar to this, the Dragino NB-IoT Shield-B5 for Arduino, although reasonably priced as an add-on, still inherits the higher expense associated with Arduino boards [43]. On the other hand, the NB-IoT Click from Mikro Elektronika exhibits versatility, allowing it to be combined with various development boards [44]. Another option worth paying attention to is the SIM7020E NB-IoT Module, specifically designed for Raspberry Pi Pico [45]. The combined cost of the two amounts to approximately 280 NOK, which is a highly competitive price point. This combination provides a product that fulfills all the system requirements and serves as a potential competitor. Taking one of the development boards with Wi-Fi, for example, the ESP32-DevKitC V4, and incorporating NB-IoT Click, results in a combined cost of around 750 NOK.

Existing NB-IoT add-on components for development boards March 2023					
Add-on component:	**Manufacturer:**	**NB-IoT:**	**Arduino C compatible:**	**Low-power:**	**Price in NOK:**
Dragino NB-IoT Shield-B5 for Arduino	Seed Studio	BC95-B20	✓	✓	390 [43]
LTE CAT M1/NB-IoT Shield for Arduino	SpakrFun	SARA-R410M-02B	✓	✓	985 [42]
NB-IoT Click	MikroE	BC95-G	✓	✓	639 [44]
SIM7020E NB-IoT Module For Raspberry Pi Pico	Waveshare	SIM7020E	✓	✓	217 [45]
Wappsto:bit NB IoT for micro:bit	Wappsto	BC66-NA	–	✓	806 [46]

Table 4: NB-IoT Add-on Components such as clicks and shields for development boards

Compared to the NB-IoT add-ons, the Wi-Fi add-ons, as presented in Table 5, exhibit more favourable pricing. The SparkFun WiFi Shield - ESP8266 is available for around 200 NOK [47], the WIFI 7 CLICK for 300 NOK [48], and the WIFI ESP CLICK for 160 NOK [49]. These options demonstrate minimal differentiation, particularly as they all incorporate parts from the ESP series.

The Wappsto:bit NB-IoT for micro:bit [46], priced at around 800 NOK, appears relatively expensive, considering it offers both narrowband and Wi-Fi capabilities. However, it is essential to highlight that this add-on is exclusively compatible with the micro:bit microcontroller board, which can be purchased for approximately 170 NOK [50]. Consequently, the combined price totals 870 NOK, still significantly higher than, for example, the LILYGO T-SIM7080G-S3, priced at 320 NOK. Additionally, the Wappsto:bit NB IoT is not intended for programming with Arduino C.

Existing Wi-Fi add-on components for development boards March 2023					
Add-on component:	**Manufacturer:**	**Wi-Fi:**	**Arduino C compatible:**	**Low-power:**	**Price in NOK:**
SparkFun WiFi Shield - ESP8266	SparkFun	ESP8266	✓	✓	197 [47]
Wappsto:bit NB IoT for micro:bit	Wappsto	ESP32	–	✓	806 [46]
WIFI 7 CLICK	MikroE	ATWINC1510-MR210PB	✓	✓	308 [48]
WIFI ESP CLICK	MicroE	ESP8266	✓	✓	159 [49]

Table 5: Wi-Fi Add-on Components such as clicks and shields for development boards

Considering all the factors, the CoT DevKit IoT project has drawn inspiration from the boards mentioned above. The primary contenders in today's market are the LILYGO T-SIM7080G-S3 and the Raspberry Pi Pico W in conjunction with the SIM7020E NB-IoT Module add-on. However, the LILYGO T-SIM7080G-S3 faces challenges due to the lack of comprehensive documentation and reliable distributors. Additionally, the combination of the Raspberry Pi Pico W and the SIM7020E NB-IoT Module does not fulfil the desired objective of integrating all functionalities onto a single board.

Upon analysing the various development boards and add-ons, it becomes evident that it is more convenient and cost-effective to opt for a microcontroller with integrated Wi-Fi capabilities. This approach not only reduces the overall expenses but also minimises physical space requirements on the final printed circuit board, making it a more practical choice. As a result, the two primary components of the system will consist of a microcontroller with an integrated Wi-Fi modem and a separate NB-IoT modem.

3.4.2 Choice of microcontroller with Wi-Fi

The choice of microcontroller with an integrated Wi-Fi modem is based on what exciting boards use, as discussed earlier, but also further market analysis of other available microcontrollers with Wi-Fi capabilities. The relevant choices are depicted in Table 6.

Microcontrollers with integrated Wi-Fi March 2023					
Microcontroller:	**Manufacturer:**	**Arduino C compatible:**	**Low-power:**	**Other capabilities:**	**Price in NOK:**
ESP32-S2-SOLO-2 module	Espressif Systems	✓	✓	–	28 [51]
ESP32-WROOM-32E module	Espressif Systems	✓	✓	Bluetooth	52 [52]
CC3220SF	Texsas Instruments	Not entirely	✓	–	70 [53]
WFI32E01	Microchip Technology	Not entirely	✓	–	145 [54]

Table 6: Available microcontrollers with integrated Wi-Fi taken into consideration

It is important to note that the ESP32-S2-SOLO-2 [51] and ESP32-WROOM-32E [40]are modules rather than solely system-on-chips (SoCs) like the CC3220SF [53] and WFI32E01 [54]. This implies that they are situated on a small circuit board primarily containing an antenna. This aspect is favourable to the ESP series, as it simplifies the design process of the eventual CoT DevKit IoT printed circuit board. Besides being modules, they are more cost-effective than the other alternatives.

All of these microcontrollers offer various low-power modes and functionalities. The ESP32-WROOM-32E also incorporates Bluetooth, which is not a requirement for this project but could be of interest for further development on the IoT development board following the completion of this thesis. In terms of Arduino C compatibility, the ESP series provides multiple libraries and references and is widely recognised for its usage. Conversely, the CC3220SF and WFI32E01 come with references for programming them in embedded C, which is the industry standard for most microcontrollers. An Arduino library would therefore, therefore, be necessary. While this is a possibility, it might be overly ambitious for this bachelor project considering time constraints. Therefore, due to these factors and their higher prices, the CC3220SF and WFI32E01PC are excluded from further consideration for implementation in this project.

The choice then lies between the ESP32-S2-SOLO-2 and ESP32-WROOM-32E. A careful examination of their datasheets reveals several key differences. As mentioned, the ESP32-WROOM-32E also incorporates Bluetooth. Furthermore, it features a dual-core CPU, unlike the ESP32-S2-SOLO-2. A dual-core CPU can offer significantly improved performance compared to a single-core CPU operating at the same speed. Multiple cores make microcontrollers able to execute multiple processes at the same. In terms of memory, the ESP32-S2-SOLO-2 has 320KB of SPRAM (static random access memory), whereas the ESP32-WROOM-32E offers 520 KB. SPRAM retains data bits in its memory as long as power is supplied, eliminating the need for continuous refreshing like dynamic RAM (DRAM) [55]. This results in improved performance and lower power consumption. From a student project perspective, greater SPRAM capacity enables the storage of more data before, for instance, transmitting it to a PC for display purposes. Moreover, the ESP32-S2-SOLO-2 provides more general input/output pins (GPIOs), as well as an LCD and camera interface, which are advantageous for certain projects. However, it should be noted that the ESP32-WROOM-32E's pins can be configured to emulate these interfaces, making them non-essential. The ESP32-S2-SOLO-2 surpasses the ESP32-WROOM-32E and is more energy efficient. When the Wi-Fi modem enters sleep mode, the ESP32-WROOM-32E consumes approximately 27mA-44mA under 160 MHz testing conditions, whereas the ESP32-S2-SOLO-2 only requires 16mA.

Nevertheless, after consulting with colleagues at Company of Things, the ESP32-WROOM-32E has been selected for this project. More specifically, the ESP32 DevKitC V4 board has the module, making it easier to implement it in a proof-of-concept. It costs 110 NOK [56]. The presence of a dual-core CPU, particularly the potential for future expansion of the IoT development board's functionalities with Bluetooth, outweighs the slightly higher power consumption and a marginal increase in cost.

3.4.3 ESP32-WROOM-32E specifications

Figure 11 depicts the ESP32-WROOM-32E module [57], while Figure 12 displays the ESP32 DevKitC V4 [58]. Although the module itself has been selected for the CoT DevKit IoT, the development board, as mentioned, has been utilised to construct the proof-of-concept in this project. Henceforth, both will be collectively referred to as "ESP32" within this report. As established, the ESP32, manufactured by Espressif Systems, serves as a suitable microcontroller module for entry-level IoT projects due to its versatile characteristics and affordable price. Table 7 presents several pertinent specifications of the ESP32. Block diagram can be found in Appendix A.6.

Figure 11: ESP32-WROOM-32E [57]

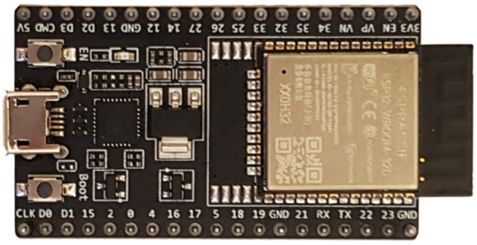


Figure 12: ESP32 DevKitC V4

ESP32 specifications:	
Number of cores	2x LX6 microprocessor
Architecture	32-bit
CPU frequency	160 MHz
Wi-Fi	802.11b/g/n/e/i
Bluetooth	v4.2 BR/EDR and BLE
Bandwidth	72 MHz
Data rate	150 Mbps
RAM	512 KB
Flash	15 MB
Power supply voltage	3.6V
GPIO pins	36
Busses	SPI, I^{2}C, UART, I^{2}S, CAN
ADC pins	18
DAC pins	2
Power save modes	Modem, Light, Deep, Hibernation

Table 7: ESP32 specifications [57]

3.4.4 Choice of NB-IoT modem

The choice of narrowband modem is based on what existing boards use, as discussed earlier, but also further market analysis of other available NB-IoT modems. The relevant choices are depicted in Table 8.

NB-IoT modems March 2023				
NB-IoT Modem:	**Manufacturer:**	**Low-power:**	**Other capabilities:**	**Price in NOK:**
BC66-NA	Quetel	✓	–	118 [59]
BC95-GV	Quectel	✓	–	147 [60]
EXS82-W	Telit Cinterion	✓	LTE-M (4G)	295 [61]
LBAD0ZZ1SE-743	Murata Electronics	✓	LTE-M (4G)	600 [62]
Monarch 2 GM02S	Sequans	✓	LTE-M (4G) and 5G ready	229 [63]
SARA-R412M-02B	u-blox	✓	LTE-M (4G)	710 [64]
SIM7022	SIMCom	✓	–	130 [65]
TX82	Telit Cinterion	✓	LTE-M (4G) and 5G ready	299 [66]

Table 8: Available NB-IoT modems taken into consideration

Firstly, SARA-R412M-02B priced at 710 NOK [64] (used by Arduino MKR NB 1500) and LBAD0ZZ1SE-743 priced at 600 NOK [62], are significantly more expensive than the rest and are therefore excluded from further consideration.

Modems from the BC-series by Quetel are used by quite a few add-ons and are priced under 200 NOK [59] [60], making them a relevant choice. Similarly, the SIM7022 module [65], an upgraded edition of the LILYGO T-SIM7080G-S3 modem, also falls within this price range. However, the other modules on the list offer cellular LTE-M/4G functionalities, which could be advantageous for future expansions of the CoT DevKiT IoT's capabilities. Making it an even more well-rounded IoT board. The relevance of this aspect was discussed with Company of Things, and if the cost increase was reasonable, it was deemed worthwhile to choose a modem that allows for this future development without the need for an entirely new board with new components. Modules such as EXS82-W [61], Monarch 2 GM02S [63], and TX82 [66], priced around 230-300 NOK, incur an additional cost of approximately 100 NOK compared to those without LTE-M functionality. This cost increment is certainly justified for Company of Things. Since the selection between these modules is motivated by the board's future evolution, it is worth noting that the hardware of Monarch 2 GM02S and TX82 is designed to be compatible with the LTE-M and NB-IoT networks' 5G specifications. Hence, the choice narrows down to these two.

Although these two modules share many similarities, they also exhibit a few differences. Both modules operate on the LTE Cat M1/NB1/NB2 bands, but the TX82 module also supports 2G. However, 2G technology is considered outdated and not relevant for Norwegian education. Both modules are capable of transmitting data, but the Monarch 2 GM02S module additionally supports SMS transmission, but this is only relevant for specific projects. On the other hand, the integration of GPS (Global Positioning System) would be a valuable future enhancement. The TX82 module has integrated GNSS (Global Navigation Satellite System) support, simplifying the connection of a GPS module. However, wanting to add more and more possible capabilities takes away from the actual project that is being created through this thesis. Company of Things agreed that easier integration with GPS is not the biggest priority. Monarch 2 GM02S module features its own LTE-M/NB-IoT protocol stack, resulting in easier management. Moreover, the Monarch 2 GM02S module offers a few more UARTs, interfaces, and GPIOs.

Cost is a crucial factor to consider, both in terms of the module itself and the evaluation kit. The Monarch 2 GM02S module is priced at 229 NOK, whereas the TX82 module costs 299 NOK, making Monarch 2 GM02S module more affordable. Since this project is a proof-of-concept, utilising evaluation kits would be ideal. Not having an evaluation kit would require creating a separate printed circuit board solely for testing and integration with other components, which can be time-consuming. The Monarch 2 GM02S NEKTAR Evaluation Kit is priced at 1218 NOK, while the TX82-DEVKIT-GEM costs 1148 NOK. Therefore, both the Monarch 2 GM02S module and its evaluation kit are cheaper than those of the TX82 module.

Considering all these factors, the Monarch 2 GM02S module emerges as the best choice. Although the easier implementation of GPS with the TX82 module is tempting, the Monarch 2 GM02S module's lower cost, LTE-M/NB-IoT protocol stack, and additional interfaces outweigh this little advantage.

3.4.5 Monarch 2 GM02S specifications

The Monarch 2 GM02S module [63] is illustrated in Figure 13, while Figure 14 displays its NEKTAR Evaluation Kit [67]. The module has been selected for implementation on CoT DevKit IoT, while the evaluation kit has been utilised to develop the proof-of-concept in this project. Henceforth, both the module and the evaluation kit will be collectively referred to as "Monarch 2" in this report. As previously stated, the Monarch 2 module is designed for cellular LTE-M/NB-IoT applications and is based on the SQN3430 chipset developed by Sequans. Its exceptional suitability for IoT projects lies, among other things, in its low power consumption, many interfaces, and ability to operate across all GSM (Global System for Mobile Communications) bands worldwide. For further details regarding its pertinent specifications, refer to Table 9. Additionally, the block diagram can be found in Appendix A.7.

Figure 13: Monarch 2 GM02S [63]

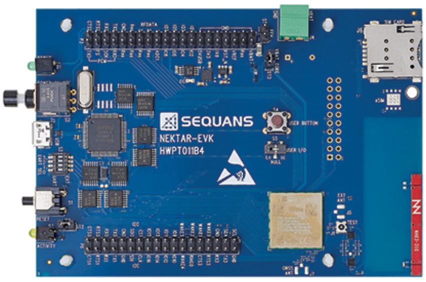


Figure 14: Monarch 2 GM02S NEKTAR Evaluation Kit [67]

Monarch 2 specifications:	
LTE features	• 3GPP LTE Release 14/15 Cat M1/NB1/NB2 compliant • LTE Cat M1: 1.1 Mbps / 0.3 Mbps UL/DL throughput • LTE Cat NB1: 62.5 kbps / 27.2 kbps UL/DL throughput • LTE Cat NB2: 160 kbps / 120.7 kbps UL/DL throughput
Single-SKU with support for LTE bands	1, 2, 3, 4, 5, 8, 12, 13, 14, 17, 18, 19, 20, 25, 26, 28, 66, 70, 71, 85
SMS	Text and PDU modes
Max transmit power	+23dBm
Single power supply	2,2-5,5V
Busses	4x UART, JTAG, I^{2}C, SPI, 2x USIM
GPIO pins	33
ADC pins	1
Software	• Field proven LTE-M and NB-IoT LTE software stack • Rich set of AT commands compatible with previous generation • IP and non-IP data delivery • LPP and certified LWM2M stack • Cloud Connector for direct HTTPS, MQTTS, CoAP to connect to all cloud platforms
Power save modes	• Power Saving Mode (PSM) • extended Discontinuous Reception (eDRX)

Table 9: Monarch 2 specifications B.2

3.5 Testing methods

The test plan for this project encompassed both integration testing and unit testing. Integration testing aimed to verify the proper functioning of different system components or modules when integrated [68]. On the other hand, unit testing focused on testing individual codes in isolation to ensure their correct functionality at an individual level [69].

The implementation process and testing conducted in this project followed the following sequence is shown below. This is done and elaborated on in Chapter "5.2 Testing and validation of final Demonstration System".

1. Initial testing of the Monarch 2 module as an individual component.
2. Integration testing between the main components, namely ESP32 and Monarch 2, to ensure their compatibility and proper communication.
3. Unit testing of various NB-IoT functionalities to validate their correct operation.
4. Unit testing of various Wi-Fi functionalities to ensure their proper functioning.
5. Unit testing of low-power mode for Monarch 2 to verify its effectiveness in reducing power consumption.
6. Unit testing of power-saving techniques for the ESP32 to assess its impact on power consumption.
7. Final comprehensive testing of the complete system, including integration testing of all hardware components and code, followed by testing all intended functionalities to ensure they were functioning as expected inside and outside. For outside testing, Tømmerdalen in Trondheim was selected, due to it being close to where the Telenor map showcases no coverage.

By following this sequence of implementation and conducting various tests, the project aimed to ensure the proper functioning and compatibility of the system components and validate the functionality of the implemented features.

4 System design

This chapter describes the realised CoT DevKit IoT proof-of-concept and its demonstration setup. Firstly, an overview of the system architecture is given and after that delves deeper into how the hardware and software works. Lastly, the reasoning behind some of the design choices are discussed.

4.1 System architecture and setup

This project designed a proof-of-concept for the CoT DevKit IoT development board. The primary objective was to establish communication and data exchange between the two main components, namely the ESP32 and Monarch 2. Enabling ESP32 to connect to the Wi-Fi network and Monarch 2 to the NB-IoT network. These are the foundational features of CoT DevKit IoT, providing students with a starting point for creating their own projects. In order to showcase the implementation of these functionalities and a demonstration system was devised. In short, the demonstration system comprises two primary buttons. The first button demonstrated Wi-Fi connectivity and the other narrowband. First the fundamental system architecture is examined and then the demo system.

4.1.1 System architecture of CoT DevKit IoT

Figure 15 illustrates the foundational system architecture of CoT DevKit IoT. ESP32 serves as the primary microcontroller, exerting control over the entire system. Additionally, it integrates a Wi-Fi modem, facilitating connectivity to the Wi-Fi network. Monarch 2 functions as the narrowband modem responsible for connecting to the NB-IoT network, with ESP32 as its host MCU (microcontroller unit). Communication between the two components is established over a UART line (Universal Asynchronous Receiver/Transmitter).

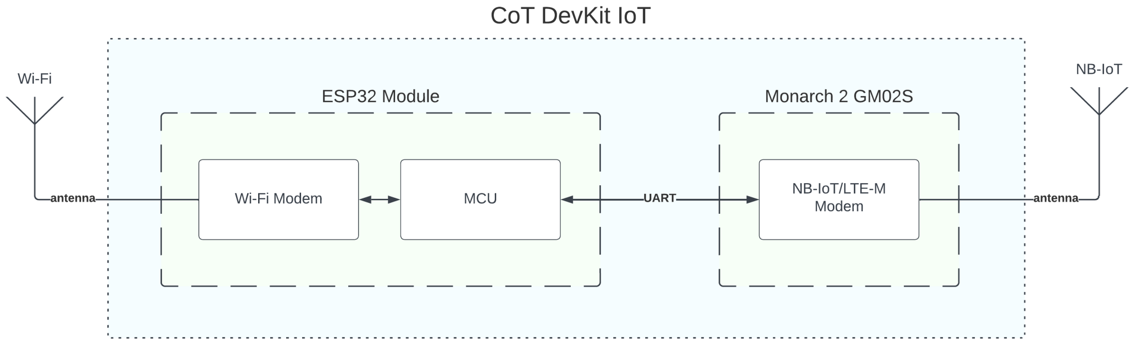


Figure 15: System architecture CoT DevKit IoT

4.1.2 System architecture with Demonstration System

Figure 16 expands upon the previous diagram and encompasses the components of the demonstration system. This system serves as a practical example to showcase the board's capability to establish connections with both networks and facilitating data exchange.

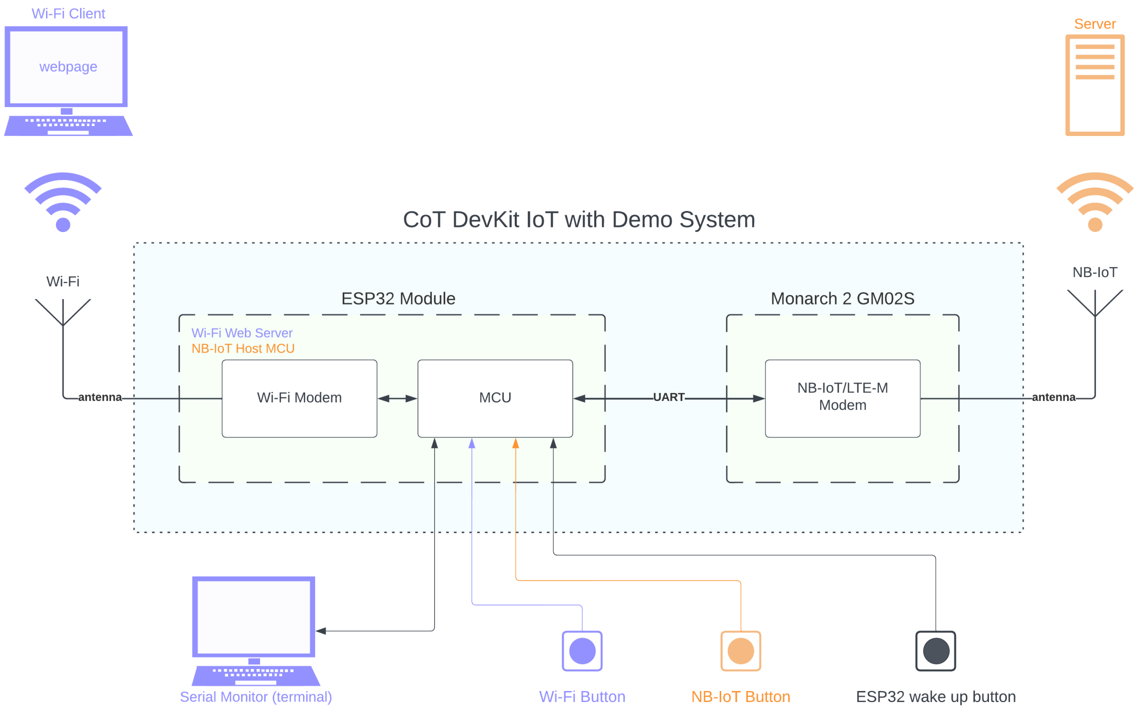


Figure 16: System architecture with demonstration system

Figure 17 depicts a logic diagram of the system functionalities. The Wi-Fi element, depicted in purple, involves configuring the ESP32 as a web server that hosts a webpage. When the "Wi-Fi Button" is pressed, the state of the button is updated on the webpage, thereby demonstrating the Wi-Fi connectivity.

The narrowband aspect, illustrated in orange, encompasses the general setup, configurations, and initialisation of Monarch 2. At system startup, the basic configurations are automatically implemented. Additionally, users have the option to manually send commands. The "NB-IoT Button" can be pressed to initiate a test signal (PING) to a designated URL address of a website, confirming the system's successful connection to the narrowband network. Monarch 2 is also in sleep mode to save power before the button is pressed and wakes the modem up.

After a predefined period of inactivity, during which neither button has been pressed, the ESP32 enters a light sleep mode to conserve power. To awaken the microcontroller unit, the "ESP32 wake-up button" is pressed.

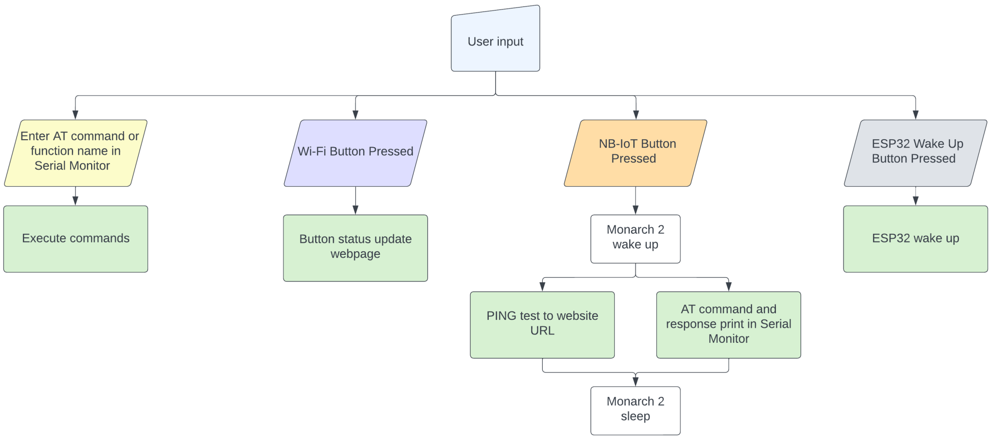

User input
Enter AT command or function name in Serial Monitor
Wi-Fi Button Pressed
NB-IoT Button Pressed
ESP32 Wake Up Button Pressed
Execute commands
Button status update webpage
Monarch 2 wake up
ESP32 wake up
PING test to website URL
AT command and response print in Serial Monitor
Monarch 2 sleep

Figure 17: System architecture with demonstration system

4.1.3 Finalised setup

The three figures below display the final setup of this project's system, meaning CoT DevKit IoT proof-of-concept with the Demonstration setup. Figure 18 shows the physical hardware setup. Note that in the image, the PC and USB cables are not included. Figure 19 is the webpage hosted by ESP32. Figure 20 is the Arduino IDE with the Serial Monitor setup, working as the primary user interface. The next chapter will explain how it works.

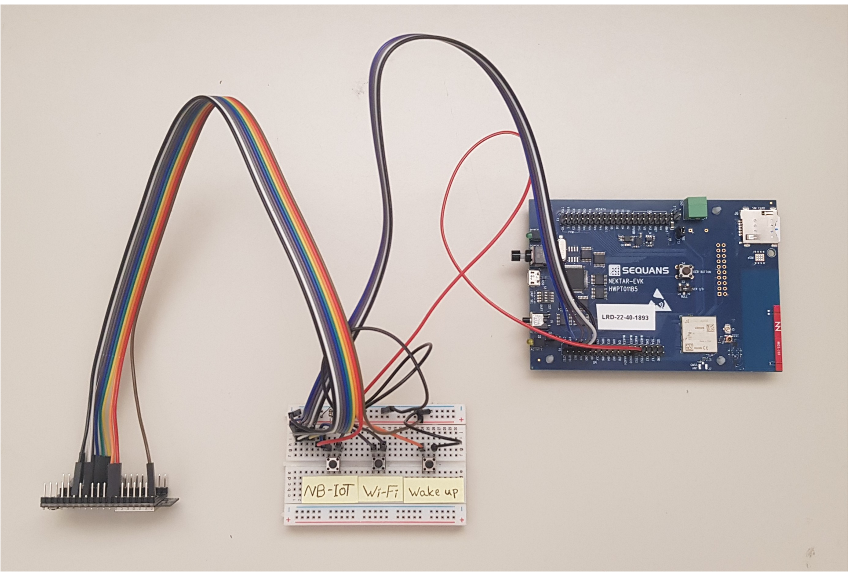
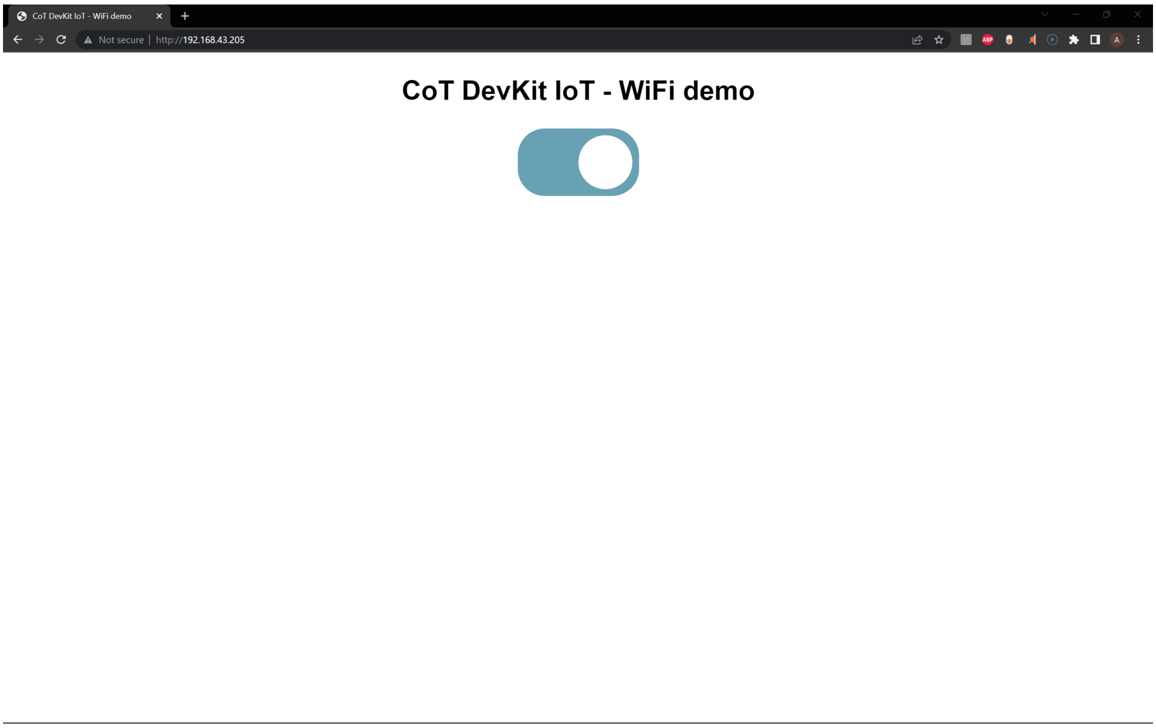
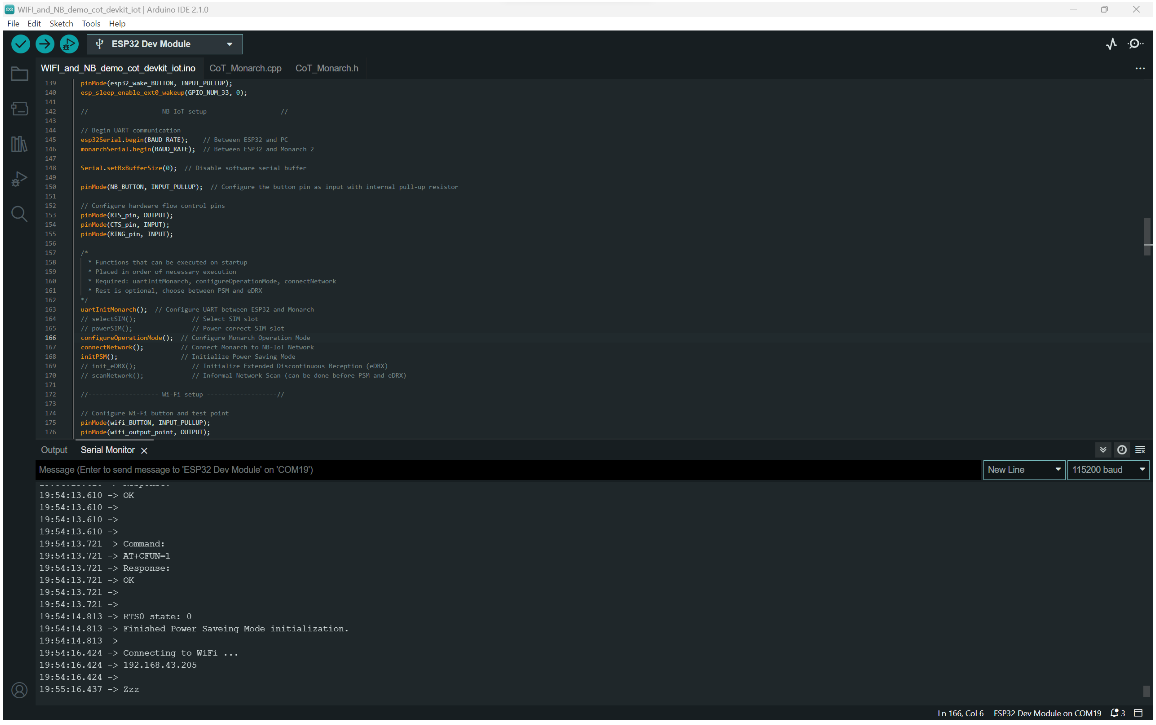


Figure 18: Physical setup of CoT DevKit IoT proof-of-concept with Demo Setup


CoT DevKit IoT - WiFi demo
Not secure | http://192.168.43.205

CoT DevKit IoT - WiFi demo

Figure 19: Webpage hosted by ESP32


WIFI_and_NB_demo_cot_devkit_iot | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 166, Col 6
ESP32 Dev Module on COM19

```
pinMode(esp32_wake_BUTTON, INPUT_PULLUP);
esp_sleep_enable_ext0_wakeup(GPIO_NUM_33, 0);

//------------------- NB-IoT setup -------------------//

// Begin UART communication
esp32Serial.begin(BAUD_RATE);    // Between ESP32 and PC
monarchSerial.begin(BAUD_RATE);  // Between ESP32 and Monarch 2

Serial.setRxBufferSize(0);  // Disable software serial buffer

pinMode(NB_BUTTON, INPUT_PULLUP);  // Configure the button pin as input with internal pull-up resistor

// Configure hardware flow control pins
pinMode(RTS_pin, OUTPUT);
pinMode(CTS_pin, INPUT);
pinMode(RING_pin, INPUT);

/*
 * Functions that can be executed on startup
 * Placed in order of necessary execution
 * Required: uartInitMonarch, configureOperationMode, connectNetwork
 * Rest is optional, choose between PSM and eDRX
*/
uartInitMonarch();  // Configure UART between ESP32 and Monarch
// selectSIM();              // Select SIM slot
// powerSIM();               // Power correct SIM slot
configureOperationMode();  // Configure Monarch Operation Mode
connectNetwork();          // Connect Monarch to NB-IoT Network
initPSM();                 // Initialize Power Saving Mode
// init_eDRX();              // Initialize Extended Discontinuous Reception (eDRX)
// scanNetwork();            // Informal Network Scan (can be done before PSM and eDRX)

//------------------- Wi-Fi setup -------------------//

// Configure Wi-Fi button and test point
pinMode(wifi_BUTTON, INPUT_PULLUP);
pinMode(wifi_output_point, OUTPUT);
```

```
19:54:13.610 -> OK
19:54:13.610 ->
19:54:13.610 ->
19:54:13.610 ->
19:54:13.721 -> Command:
19:54:13.721 -> AT+CFUN=1
19:54:13.721 -> Response:
19:54:13.721 -> OK
19:54:13.721 ->
19:54:13.721 ->
19:54:14.813 -> RTS0 state: 0
19:54:14.813 -> Finished Power Saveing Mode initialization.
19:54:14.813 ->
19:54:16.424 -> Connecting to WiFi ...
19:54:16.424 -> 192.168.43.205
19:54:16.424 ->
19:55:16.437 -> Zzz
```

Figure 20: Arduino IDE with Serial Monitor (dark mode)

4.2 Detailed system design: Hardware

As previously stated, the ESP32 assumes the role of the master, governing the entire system, while Monarch 2 acts as its subordinate. The ESP32 exclusively manages the Wi-Fi functionality, whereas the NB-IoT functionality naturally encompasses both modules. A block diagram of pin connection for the system is depicted in Figure 21. Schematics and pinout diagram for ESP32 can be found in Appendix A.8 and A.4. Schematics and pinout for Monarch 2 can be found in Appendix A.10 and A.5. Figure 18 above shows the physical hardware realisation of the main and final setup for this project report.

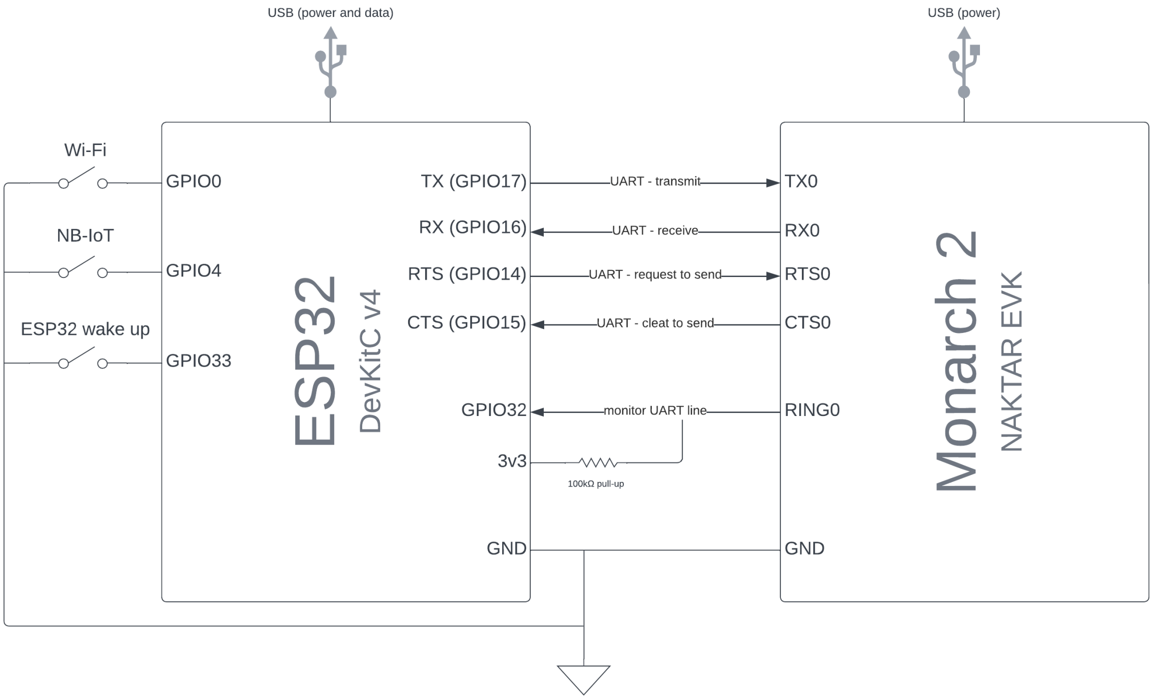


Figure 21: Block diagram with system connections

The system is powered through USB cables connected to a PC. The cable connected to Monarch 2 serves solely as a power source, whereas the cable connected to the ESP32 serves the dual purpose of power supply and monitoring/displaying transmissions and responses. Buttons, resistors, and all cables are interconnected using a breadboard. To establish a connection to the NB-IoT network, a SIM card (subscriber identity module) from Telenor is inserted.

The ESP32 controls Monarch 2 by transmitting AT commands over the UART line. AT commands, also known as "Attention" commands, are specifically designed for modem management. For further details, please refer to Chapter "2.4.1 AT commands". The UART line also enables data transmission between the ESP32 and Monarch 2. The selected UART configuration for this communication is "Type 1," as described in Monarch 2's System Integration Guide (B.5 page 1). This configuration was chosen due to its support for hardware flow control and low-power capabilities. The UART communication takes place between ESP32's UART2 pins and Monarch 2's UART0 pins, utilising the DCE-DTE convention (see below) with hardware flow control (B.6 page 13). The serial link settings for this communication include a baud rate of 115200, 8 data bits, no parity, and 1 stop bit (B.6 page 14). Note that "0" on the "UART SEL" (S3) switch on the Monarch module must be changed to "EXT". Further information regarding this can be found in Chapter 5.1.2.

4.2.1 DCE-DTE connection

As mentioned, this system utilises the DCE-DTE convention. Where DTE refers to "Data Terminal Equipment," representing a user device, and DCE refers to "Data Communications Equipment," representing a network device [70]. Within this context, the ESP32 serves as the DTE, functioning as the master of the system, while Monarch 2 is the DCE and serves as the subordinate. Acting as the DTE, the ESP32 serves as both the source and destination for data transmission. Conversely, as the DCE, Monarch 2 is responsible for the transmission and reception of data over the narrowband network.

In a conventional DTE-DTE connection, the transmit pin (TX) on one device is linked to the receive pin (RX) on the other device, while the receive pin (RX) is connected to the transmit pin (TX) [71]. However, in the case of a DCE-DTE connection, the DCE perceives data from the same perspective as the DTE. Consequently, the DCE (Monarch 2) connects its TX pin to the TX pin of the DTE (ESP32) and its RX pin to the RX pin of the DTE.

4.2.2 Hardware flow control

Hardware flow control was implemented to regulate data flow between ESP32 and Monarch 2. This mechanism's primary objective is to prevent data loss and buffer overflow by ensuring that data transmission occurs only when the receiving device is ready to accept data. This was done by employing Request to Send (RTS) and Clear to Send (CTS) signals and pins on both devices [71].

When the ESP32 wants to send data, it toggles the RTS signals to indicate this. Monarch 2 receives this and toggles its CTS pin low, signifying that it is ready to receive data or AT commands (B.5 page 5). If the buffer is full, the data transmission is temporarily halted until ready and then toggles CTS.

Monarch 2 also allows for an additional flow control pin. The RING line monitors pending data and unsolicited result codes (URC) on the UART line. An unsolicited result code is a string message that is not triggered as an information text response to a previous AT command and can be output at any time to inform a specific event or status change [72]. This is, for example a "+SYSSTART" response from Monarch 2 when it is powered up.

4.2.3 Web server

The Wi-Fi element only uses the ESP32. The ESP32 offers three predefined Wi-Fi modes: Access Point, Station, and both modes simultaneously [73]. In Wi-Fi networks, access points provide network connectivity, while stations are devices that connect to these access points. For example, a Wi-Fi router in a home acts as an access point and a mobile phone functions as a station. In this project, the ESP32 operates as an access point, establishing a connection with a Wi-Fi router and creating its own Wi-Fi network for station clients to connect to. Figure 76 illustrates this setup.

Technically, ESP32 acts as a "soft (software) access point" since it does not connect further to a wired network [74]. In this project, the ESP32 operates as a web server, hosting a webpage. As a web server, it stores, processes, and delivers webpage content using the Hypertext Transfer Protocol (HTTP) [75]. HTTP serves as a method for encoding and transporting information between the client and the web server [76]. The content of the webpage is created using Hyper Text Markup Language (HTML) code.

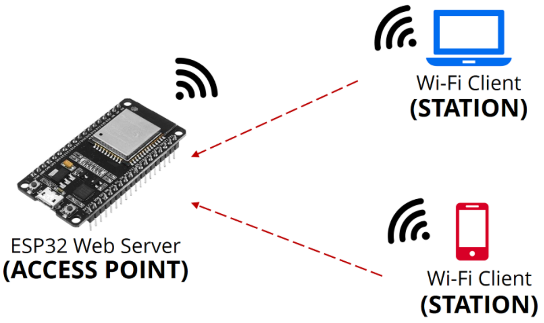


Figure 22: ESP32 as access point and web server [73]

4.2.4 ESP32 low-power

The ESP32 microcontroller is highly capable but can, at times, be relatively power-hungry. To address this issue, the ESP32 offers various power-saving modes. The two major beings "Light-sleep" and "Deep-sleep", but they also supports a few sup-modes [77]. When the MCU is in "Active" mode, all functionalities of the module are activated, resulting in a power consumption ranging from approximately 160-260mA. Depicted in Figure 23 [78].

In Light-sleep mode, shown in 24, Wi-Fi and Bluetooth functionalities are turned off, the clock signal is removed or ignored when the circuit is not in use, meaning that it clock-gates the digital peripherals, CPU (Central Processing Unit, and most of the RAM (Random-Access Memory). This approach contributes to additional power savings down to 0,8mA [78]. A sub-mode, called Modem-sleep, turns only of Wi-Fi and Bluetooth but uses 3-20mA [78].

Moving on to Figure 25, the Deep-sleep mode is presented. In this mode, all digital peripherals, the CPU, and a significant portion of the RAM are disabled. Only specific components, such as the ULP (ultra-low power) Co-processor, RTC (Real Time Clock) Controller, RTC Peripherals, and RTC fast and slow memory, remain active [78]. The next sub-mode is Hibernation, where also both the internal 8 MHz oscillator and the ULP Co-processor are disabled [78]. Only one RTC timer, operating on a slow clock, and a few RTC GPIOs are kept active to facilitate waking the chip. Here no data can be saved, but a substantial power reduction of 2,5µA is achieved [78].

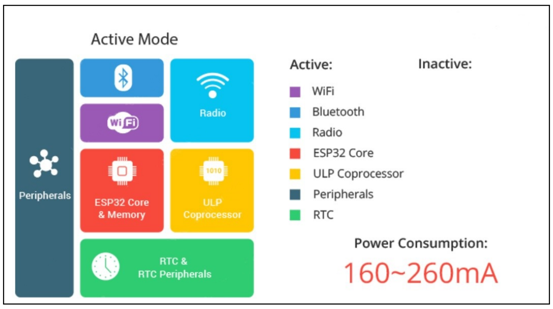


Figure 23: Active mode details [78]

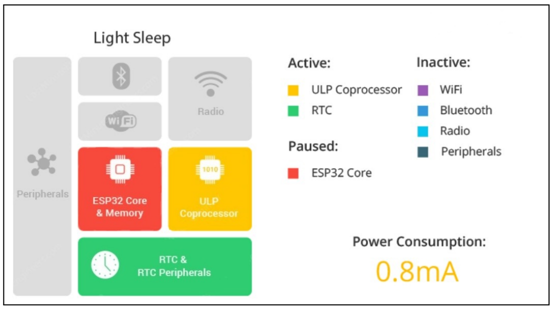


Figure 24: Light-sleep mode details [78]

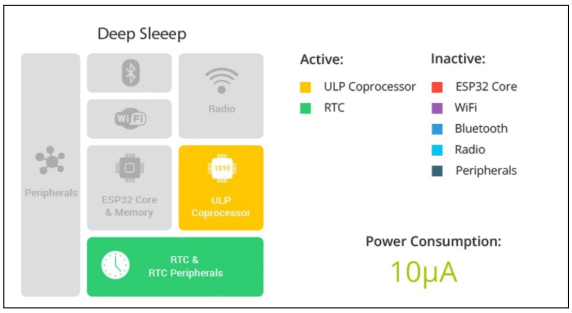


Figure 25: Deep-sleep mode details [78]

4.2.5 Monarch 2 low-power

The Monarch 2 can require very low power consumption. In the context of Monarch 2 power optimization, it is crucial to differentiate between modes and techniques. Modes represent distinct power states that offer varying functionalities, while techniques involve methods for transitioning into and out of these states. Power modes are described in Figure 27 and illustrated in Figure 26, while the power saving techniques PSM and eDRX are delved into further below.

The power modes of Monarch 2 encompass a sequential progression of functionalities. The device operates with all functionalities enabled in "Active" mode. In "Standby" mode, the network accessibility is restricted. "Sleep" and "Deep Sleep" deactivate most functionalities to achieve greater power savings. When going into sleep mode, it enters Radio Resource Control (RRC) idle, which means that it disables the connection to the NB-IoT network. In order to resume operation from the Sleep and Deep Sleep modes, the modem necessitates a wake-up mechanism. In this project, the RTS0 (request to send) pin on Monarch 2 is configured as a wake source when toggled. The system stays in the loop between "Active Standby" and "Sleep/Deep Sleep" in Figure 26, waking up whenever the "NB-IoT Button" is pressed as this signals the RTS0 and so on.

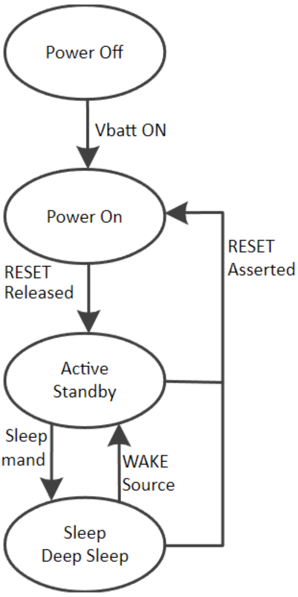
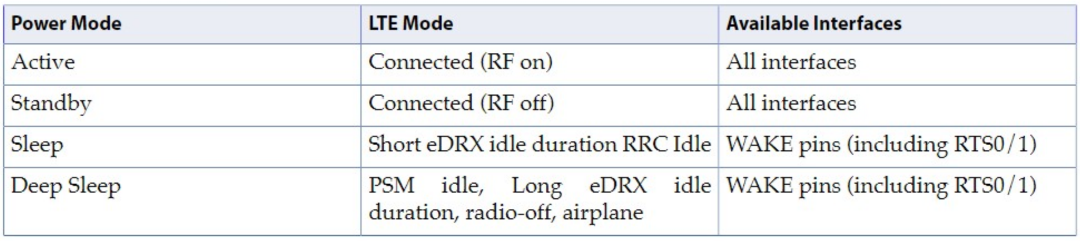


Figure 26: Modes and transitions (B.2 page 14)



Power Mode	LTE Mode	Available Interfaces
Active	Connected (RF on)	All interfaces
Standby	Connected (RF off)	All interfaces
Sleep	Short eDRX idle duration RRC Idle	WAKE pins (including RTS0/1)
Deep Sleep	PSM idle, Long eDRX idle duration, radio-off, airplane	WAKE pins (including RTS0/1)

Figure 27: Monarch 2 power modes and functionalities (B.2 page 14)

Monarch 2 incorporates two primary power reduction techniques: Power Saving Mode (PSM) and extended Discontinuous Reception (eDRX). A more fundamental explanation of these techniques can be found in Chapter 2 Theoretical framework, section 2.5 Low-power. The module power mode selection is internally managed by the module software (B.6 page 15), and both techniques have been made available in the project. In summary, these protocols allow the modem to enter sleep mode while retaining network configurations. During these states, the UART functionalities of Monarch 2 are deactivated. Timers are employed to periodically wake up and monitor the modem, although external events like pin toggling can also trigger the wake-up process (RTS0).

The distinction between PSM and eDRX lies in the frequency of monitoring and the depth of sleep achieved. PSM enables significantly longer sleep duration, making it suitable for devices that do not require continuous active interaction with the network. On the other hand, eDRX is designed for more frequent paging and achieves a shallower sleep mode compared to PSM, resulting in reduced energy consumption during wake-up. eDRX mostly enters sleep mode but can also reach deep sleep. While PSM always goes into deep sleep. In PSM mode, both the receiver (RX) and transmitter (TX) are turned off, while in eDRX, the TX is disabled, and the RX is periodically activated. Maximum expected power consumption in PSM sleep is 48µA, and eDRX sleep is 45-268 µA (B.7 page 9). The project system does not wait for sensor data or anything equivalent where the user does not activate it, only button input. Therefore, monitoring frequently with timers is not necessary, and in later testing, PSM is used. Even though, as mentioned, it is configured to also work with eDRX.

The Paging Cycle Length (PCL) determines the interval at which the network is monitored. For Monarch 2's eDRX implementation in NB-IoT, the minimum PCL is 10,24 seconds, while the maximum is 2,91 hours. Conversely, with PSM, the maximum PCL is 310 hours or approximately 13 days. A significant difference.

In Figures 28 and 29 PSM mechanism is illustrated. Figure 28 is a power signal example, and Figure 29 is a diagram from Monarch 2's datasheet explaining it. TAU stands for Tracking Area Update and is a signaling procedure to inform the network about its location to optimise resource allocation and ensure efficient communication. The "TAU+data" period describes when there is data transfer and updating of the tracking area. The duration between two TAU is determined by the T3412 Periodic TAU Timer. T3324 is the Active Timer and defines the time the device is reachable. It is reachable in the "Short idle" period in the figure, where it can monitor paging as the T3324 counts down before going into PSM sleep.

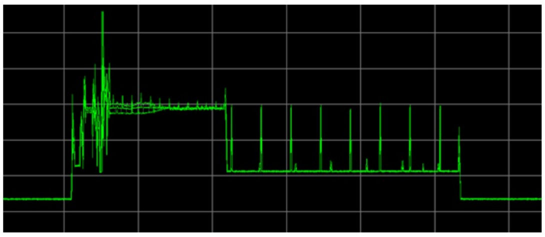


Figure 28: PSM signal example [79]

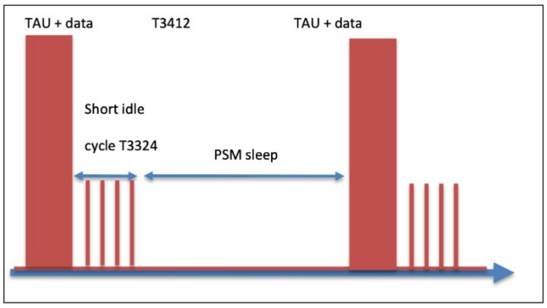


Figure 29: Monarch 2 PSM diagram (B.8 page 3)

Figure 30 and 31 illustrate eDRX mechanism. Figure 30 is a power signal example and Figure 31 is a diagram from Monarch 2's datasheet explaining it. Paging Transmission Window (PTW) defines how long it monitors. The PTW window in eDRX is when the device is reachable. eDRX also uses the T3412 Periodic TAU Timer and TAU in between a few PTWs, but is not illustrated in the figures.

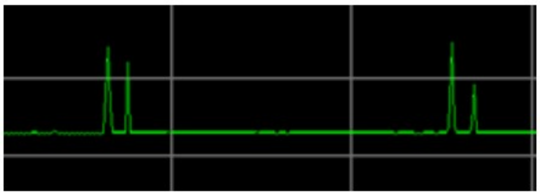


Figure 30: eDRX signal example [79]

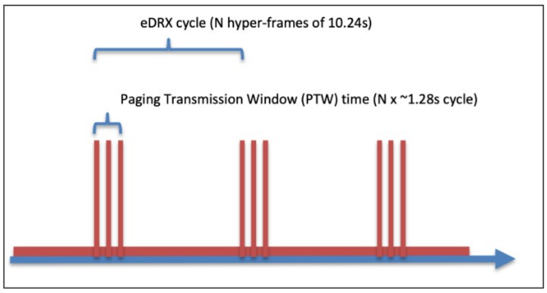

eDRX cycle (N hyper-frames of 10.24s)
Paging Transmission Window (PTW) time (N x ~1.28s cycle)

Figure 31: Monarch 2 eDRX diagram (B.8 page 3

The Telenor network supports both techniques [79]. Mentioned earlier, the demonstration system has implemented possibility to enter both PSM and eDRX. However, testing mostly used PSM.

4.3 Detailed system design: Software

The software for the Wi-Fi and NB-IoT demonstration setup was developed using Arduino C. In addition to this, a custom library was created to handle Monarch configurations, making it suitable for integration into any CoT DevKiT IoT project, not limited to the current project. The complete code is titled "Final_complete_code-WIFI_and_NB_demo_cot_devkit_iot.ino", see Attachment B.13.

The entire development process was carried out using Arduino IDE. Arduino IDE also serves as the user interface (UI), as it displays all communication and data transfer in Serial Monitor. Users have the capability to manually input AT commands or predefined function names via the Serial Monitor, enabling them to control Monarch 2 through the ESP32. The first time plugging the Monarch 2 into a PC, the necessary Sequans Communications drivers will be installed.

4.3.1 Flowchart of code

The flowchart presented in Figure 32, depicted on the subsequent page, provides a visual representation of the logical sequence of operations. The program adheres to a conventional structure of Arduino C code, featuring a "void setup()" section that executes once and a "void loop()" section that runs continuously as long as the system is powered. The rest of this sub-chapter will dive deeper into how the code works.

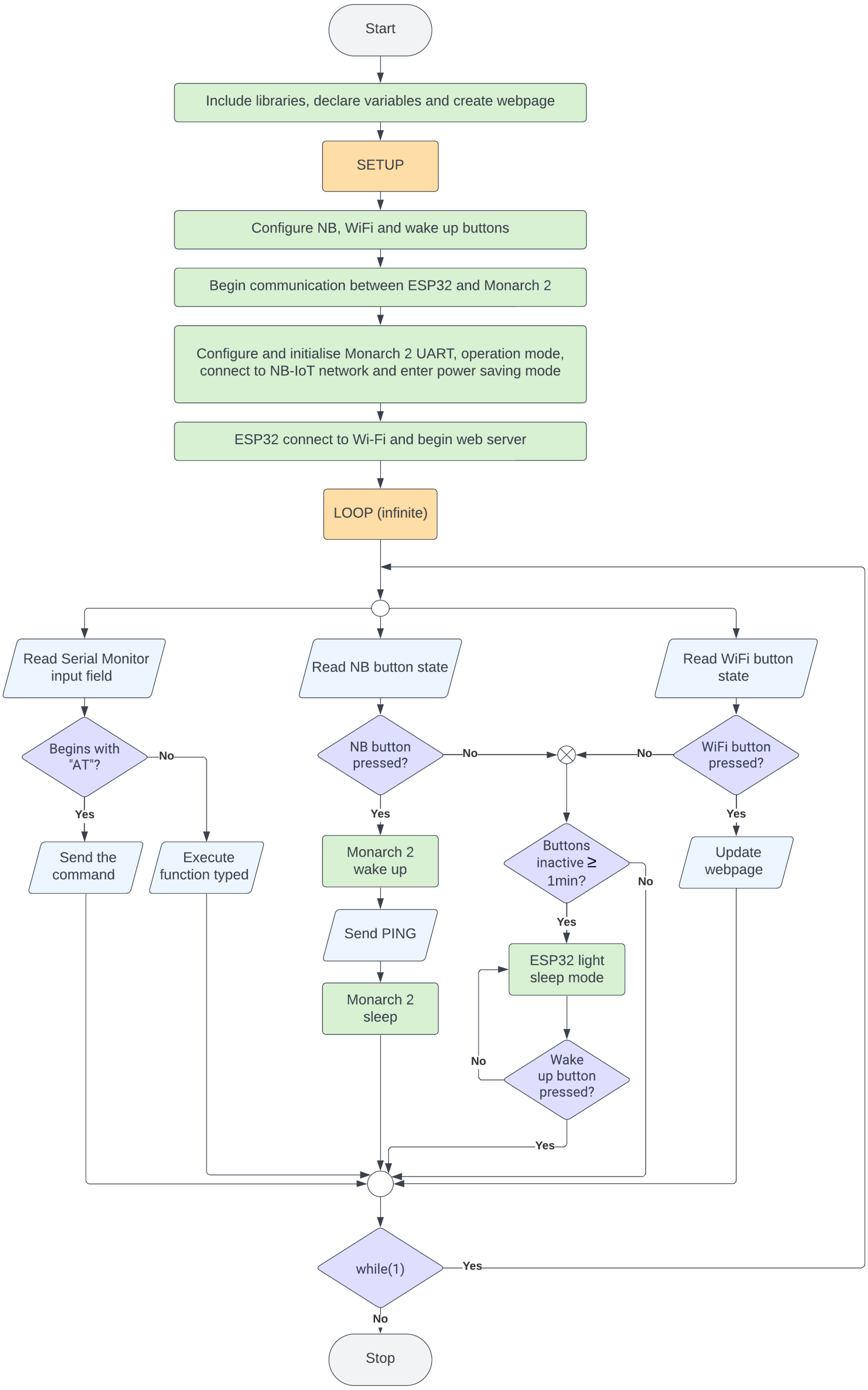


Figure 32: Flowchart of code

4.3.2 Start

The first section of the code incorporates all necessary libraries and variables, as well as webpage arrangement.

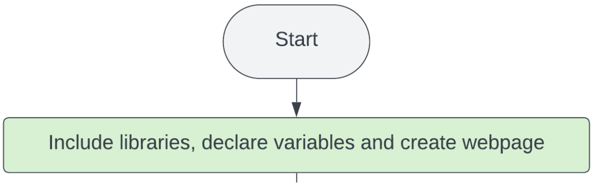


Figure 33: Start

Libraries used:

- HardwareSerial.h: This library facilitates UART communication between Monarch 2, ESP32, and the PC.

- CoT_Monarch.h: The CoT_Monarch.h library was developed as a part of this bachelor thesis and incorporates specific configurations and functionalities for Cot DevKit IoT.

- ESP32 add-on: The ESP32 add-on library must be installed in Arduino IDE to enable ESP32 development. Detailed instructions can be found in the linked guide [80]. The rest of the libraries are included in this add-on.
  - WiFi.h: The WiFi.h library provides the necessary functionalities for Wi-Fi operations on the ESP32.
  - AsyncTCP.h: The AsyncTCP.h library is used for asynchronous transmission control protocol, which is required for the ESP32 to act as a web server.
  - ESPAsyncWebServer.h: This library enables the ESP32 to function as an asynchronous web server.

CoT_Monarch.h library:

Consists of hardware UART specifications and handles the response from Monarch 2 and different AT functions that can either be placed in the code or typed manually in Serial Monitor.

The hardware specifications for the UART lines are defined from the perspective of the ESP32. For the communication between Monarch 2 and the ESP32, the UART2 pins of the ESP32 are utilised. On the other hand, the UART0 of the ESP32 is connected to the PC via the USB cable. This configuration is achieved using the HardwareSerial class, as demonstrated in the code snippet provided in Figure 34. The RX and TX pins are defined within this class, while the hardware flow control pins are defined separately, as indicated in the code.

```
// UART ports on ESP32 defined in .cpp file:
HardwareSerial monarchSerial(2); // Between ESP32 and Monarch 2
HardwareSerial esp32Serial(0); // Between ESP32 and PC (USB-cable)

// Hardware flow control pins declared in .h file:
const int RTS_pin = 14;   // Request to Send, also wake up source for Monarch 2
const int CTS_pin = 15;   // Clear to Send
const int RING_pin = 32;  // Monitors data and URC on UART line
```

Figure 34: Hardware UART specifications in CoT_Monarch library

"void handleResponse()" is a central function within the library that monitors and handles data on the ESP32-Monarch UART-line and displays the responses in Serial Monitor. If an AT function name is typed (see below), and any of the commands return "ERROR", the execution of the rest is stopped to protect the system. This function is used in all communication between the two devices.

The predefined AT functions are comprised of a combination of AT commands specific for Monarch 2 configurations. Below is a list of all of them. It is necessary to implement the UART initialisation, operation configuration mode and connect to the network in that order for the Monarch to operate. SIM should be checked if facing any connectivity problems.

Available AT command functions:

- uartInitMonarch ⇒ Configure UART on Monarch 2
- selectSIM ⇒ Select SIM slot
- powerSIM ⇒ Power correct SIM slot
- configureOperationMode ⇒ Configure Monarch Operation Mode
- connectNetwork ⇒ Connect Monarch to NB-IoT Network
- initPSM ⇒ Initialize Power Saving Mode
- init_eDRX ⇒ Initialize Extended Discontinuous Reception (eDRX)
- scanNetwork ⇒ Informal Network Scan (can be done before PSM and eDRX)

Using the "uartInitMonarch" function as an example, as shown in Figure 35, all the commands required for configuring the UART with hardware flow control and setting RTS as a wake-up source are listed. The function iterates through these commands, sending one command at a time.

```
void uartInitMonarch() {
  Serial.println("Beginning␣UART␣Monarch␣2␣init!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    "AT",                                             // Test AT command connection
    "AT+CFUN=5",                                      // Entering manufacturing mode
    "AT+SQNIPSCFG=2,100",                             // UART timeout 100ms
    "AT+SQNHWCFG=\"uart0\",\"enable\",\"rtscts\"",    // Enable hardware flow control on Monarch UART0
    "AT+SQNHWCFG=\"wakeRTS0\",\"enable\"",            // Setting RTS0 as wake source
    "AT+SQNRICFG=1,3,100",                            // RING timeout 100ms
    "AT^RESET"
  };

  // Loop through each command and forward to Monarch
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it is the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }
    delay(2000);
  }
  Serial.println("Finished␣UART␣init.");
  Serial.println();
  delay(500);
}
```

Figure 35: AT command function for Monarch 2 UART configurations

Creating webpage:
Setting up the webpage includes defining password and SSID (Service Set Identifier/-name of network) for the local Wi-Fi network that the ESP32 will use to behave as an access point. HTML (Hyper Text Markup Language) code is utilised to design the visual appearance of the webpage, which is largely inspired by the linked tutorial [81]. This code defines the layout, structure, and styling elements of the page. Port 80, which is a commonly used internet communication protocol associated with HTML, is employed for transmitting and receiving data. The asynchronous web server then stores different placeholders for states of the hardware.

4.3.3 Setup

This is the "void setup()" section that runs once when powering up the system.

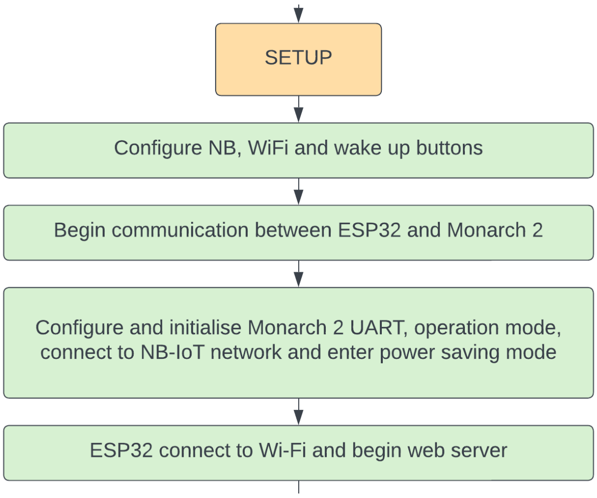

SETUP
Configure NB, WiFi and wake up buttons
Begin communication between ESP32 and Monarch 2
Configure and initialise Monarch 2 UART, operation mode, connect to NB-IoT network and enter power saving mode
ESP32 connect to Wi-Fi and begin web server

Figure 36: Setup

The NB, Wi-Fi, and wake-up buttons are configured and interrupt enabled for the ESP32 wake-up button.

First, the NB-IoT setup is established by beginning communication between ESP32 and Monarch 2. The essential AT command functions are implemented. The "initPSM" function, which initialises power saving mode, is also applied, allowing the modem to directly enter sleep mode.

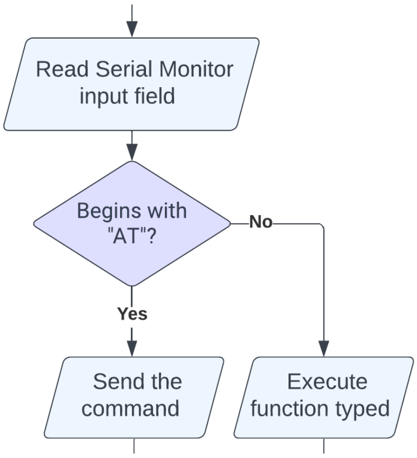
Then the Wi-Fi setup is initiated. ESP32 connects to the network and prints its IP address (Internet Protocol), which is also the URL (Uniform Resource Locator) of the webpage that the user types in a desired web browser. As a web server, it uses GET requests to set up the web page and call button data.

4.3.4 Loop

The "void loop()" runs continuously and infinitely until the power is shut off. This function is divided into three parts to examine and understand the code better.

Read Serial Monitor input field:
Figure 37 illustrates the flowchart and accompanying code for the process of entering AT commands or function names in the Serial Monitor. This code snippet encompasses three distinct functionalities. Firstly, it constantly monitors the serial line between the ESP32 and the PC, displaying any data transmitted within the system on the Serial Monitor. Secondly and thirdly, it allows the user to manually input commands or function names via the Serial Monitor input field.

When the user enters something that begins with "AT," indicating an AT command, it is sent to the NB-IoT modem for execution. On the other hand, if the input does not start with "AT" but contains some text, it is interpreted as a function name. If the function name is recognisable, it triggers the execution of a predefined function. These predefined functions are defined within the CoT_Monarch library and entail multiple essential AT commands that configure the specified functionality. The library also handles the response from Monarch 2 and prints it accordingly. In the event that any of the commands return an "ERROR" response, the execution of that function is aborted. Further details on the available functions and additional information can be found in the preceding "4.3.2 Start" and "4.3.3 Setup" within this Chapter. The "executeFunction()" is what allows the user to type in function names.



```
//----- Type AT command or Function name -----//

// Monitoring uart line
if (esp32Serial.available()) {
  // Read input typed in Serial Monitor
  String input = esp32Serial.readStringUntil('\n');
  input.trim();  // Remove whitespaces

  // Check if the input is an AT command
  if (input.startsWith("AT")) {
    // Send the AT command to Monarch
    monarchSerial.println(input);  // Forward data
    Serial.println();
    Serial.println("Command:␣");
    Serial.println(input);  // Echo
    delay(100);

    // Read and print the response from Monarch
    Serial.print("Response:␣");
    while (monarchSerial.available()) {
      char c = monarchSerial.read();
      esp32Serial.write(c);  // Forward data
    }
    Serial.println();
  }
  // Else a function name was typed
  else {
    executeFunction(input);
  }
}
```

Figure 37: Flowchart and code: Type AT command or Function name in Serial Monitor

Read NB button state:
Figure 38 presents the flowchart and corresponding code that handles the NB-IoT button and its state monitoring. The button's state is continuously observed throughout the process. When the button is pressed, the following sequence of actions occurs:

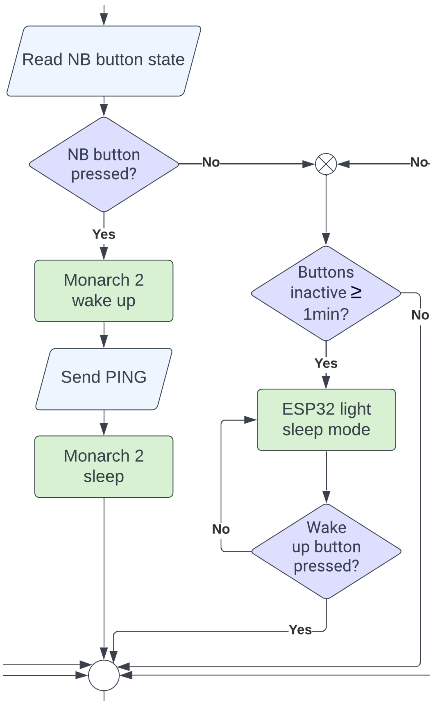
First, the RTS pin (request to send) is toggled, serving the purpose of waking up the modem. In the demonstration system, the modem is initially in power-saving mode. As stated before, the RTS pin is configured as a wake source for Monarch 2, although it still functions within hardware flow control.

Then the CTS pin (clear to send) is checked to ensure that Monarch 2 is ready for data exchange. It must be in a low state to indicate readiness. This verification is followed by the execution of the "AT+CFUN?" command, which checks the phone/cellular functionality and that it is connected to a network (B.9 page 156).

The CoT_Monarch library includes a predefined function that handles the response from "AT+CFUN?". If the response is "OK," it indicates that the system is awake. Subsequently, a PING test is sent to "www.sequans.com" to verify the connection to the NB-IoT network (B.9 page 48). This is just a random website selected. The "AT-PING=www.sequans.com" command should return the IP address of the website.

After completing the above steps, the RTS pin is toggled once again, placing the system back into sleep mode.

Furthermore, the elapsed time since the button was last pressed is measured. If neither the NB-IoT button nor the Wi-Fi button has been pressed within the last minute, the ESP32 enters light sleep mode. In this mode, the NB-IoT connection is disabled, while the configurations from the setup section are retained. For additional information, refer to "4.2.4 ESP32 low-power".



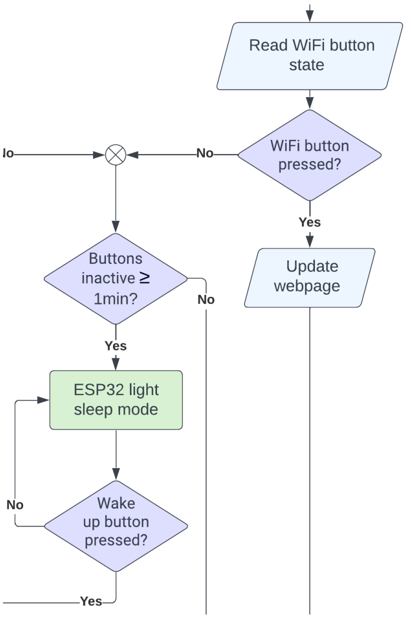
```
//-------- NB-IoT button (send PING) --------//

// Read the current button state
nb_buttonState = digitalRead(NB_BUTTON);

// Check if the button pressed
if (nb_buttonState != nb_lastButtonState &&
    nb_buttonState == LOW) {
  Serial.println("NB-IoT␣button␣pushed!");

  // Toggle RTS0 to wake modem from power save mode
  // RTS0 is configured as wake source
  Serial.println("Waking␣up␣modem");
  digitalWrite(RTS_pin, HIGH);
  int rts_state = digitalRead(RTS_pin);
  Serial.print("RTS0␣state:␣");
  Serial.println(rts_state);
  delay(100);

  // Check if CTS pin is low (indicating modem is
      ready to receive data and AT commands)
  if (digitalRead(CTS_pin) == LOW) {
    Serial.println("CTS␣is␣clear");
    String checkReady = "AT+CFUN?";
    monarchSerial.println(checkReady);
    Serial.println("Checking␣if␣buffer␣is␣ready␣for␣
        data␣and␣AT␣commands");
    Serial.println(checkReady);

    // Handle response from Monarch and send PING
        test to www.sequans.com to see that it is
        attached to NB-IoT network
    buttonNB_handleResponse_sendPING();

  } else {
    Serial.println("CTS␣not␣clear␣and␣Monarch␣2␣not␣
        ready");
  }
  delay(100);

  // Toggle RTS pin after data transmission is
      complete
  digitalWrite(RTS_pin, LOW);
  Serial.print("RTS0␣state:␣");
  Serial.println(rts_state);
  Serial.println();

  nb_lastButtonPressTime = millis(); // Keep time
  delay(100);
}

// Update button state
nb_lastButtonState = nb_buttonState;
```

Figure 38: Flowchart and code: NB-IoT button (send PING)

Read Wi-Fi button state:
Figure 39 illustrates the flowchart and accompanying code responsible for managing the Wi-Fi button and its state monitoring. The following actions occur when the button is pressed:

The program records the time since the button was last pressed. This time measurement serves two purposes: debouncing the button to eliminate noise and determining when to enter sleep mode. After debouncing the button, the updated button state is sent to and visualised on the webpage.

As previously mentioned, if neither the NB-IoT button nor the Wi-Fi button is pressed within a one-minute timeframe, the ESP32 enters light sleep mode. In this mode, Wi-Fi functionalities are disabled.



```
//----- Wi-Fi button (change state webpage) -----//

// Read the current button state
int wifi_actual_buttonState = digitalRead(wifi_BUTTON);

// If button pushed, save time
if (wifi_actual_buttonState != wifi_lastButtonState) {
  wifi_lastButtonPressTime = millis();
}
// If button pushed, change state on webpage
if ((currentTime - wifi_lastButtonPressTime) >
    debounceDelay) {
  if (wifi_actual_buttonState != wifi_buttonState) {
    wifi_buttonState = wifi_actual_buttonState;
    if (wifi_buttonState == HIGH) {
      wifi_outputState = !wifi_outputState;
    }
  }
}

// Change state on output test point
digitalWrite(wifi_output_point, wifi_outputState);
// Update button state
wifi_lastButtonState = wifi_actual_buttonState;
```

Figure 39: Flowchart and code: Wi-Fi button (change state on webpage)

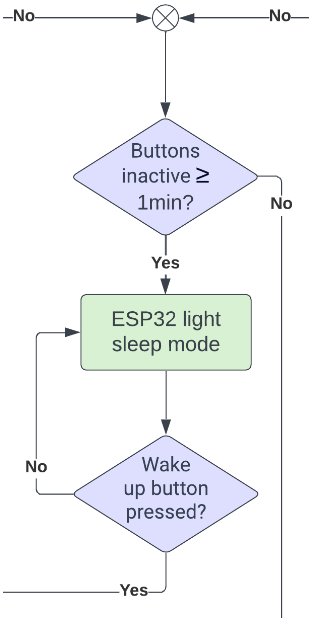
ESP32 light sleep mode

Figure 51 depicts the flowchart and corresponding code responsible for the ESP32 entering light sleep mode when the system remains inactive for over a minute. In this context, "inactive" refers to the absence of button presses for the NB-IoT and Wi-Fi buttons. When the system enters light sleep mode, the following actions take place:

Wi-Fi and UART peripherals are disabled, subsequently leading to the deactivation of NB-IoT connectivity as well. As a result, both the NB-IoT and Wi-Fi buttons become non-functional, and the user can no longer input any commands via the Serial Monitor.

However, the configurations specified in the "Setup" section are retained, enabling these functionalities to be reinstated immediately upon waking up the system by pressing the "ESP32 wake-up button". In "void setup()", the function esp_sleep_enable_ext0_wakeup(GPIO_NUM_33, 0) configures the external pin connected to the button. This pin triggers an interrupt when it is in a low state (0).

It is important to note that light sleep mode provides power-saving benefits by disabling certain components while maintaining the ability to restore functionality upon wake-up. Monarch 2 retains its configuration even though the UART line is disabled and the webpage is still configured.



```
//----- ESP32 enter Light Sleep Mode -----//

// Entering Light Sleep Mode after both buttons inactive for 1min
if ((currentTime - nb_lastButtonPressTime >= lightSleepTime) && (currentTime - wifi_lastButtonPressTime >=   lightSleepTime)) {
  Serial.println("Zzz");

  // Enter light sleep mode
  // Function is from esp_sleep.h library in ESP32 add-on
  esp_light_sleep_start();

  // Reset last button pressed times
  nb_lastButtonPressTime = currentTime;
  wifi_lastButtonPressTime = currentTime;
}
```

Figure 40: Flowchart and code: ESP32 enter Light Sleep Mode after inactivity (buttons not pressed)

4.4 Design choices

In this chapter, the rationale behind some of the design choices is discussed.

4.4.1 System architecture with Demonstration Setup

The demonstration setup and code included in this project serve to showcase the basic functionalities of CoT DevKit IoT. They aim to inspire students by illustrating how the board can be utilised. The setup demonstrates how Monarch 2 can be controlled, and data can be exchanged with ESP32. The buttons in the system simulate input data, mimicking the functionality of sensors that students can integrate into their own projects. Furthermore are, power-saving techniques incorporated to show them how that can be done.

4.4.2 Enabling Arduino IDE Serial Monitor as functional terminal

A microcontroller device connected to a PC over a UART line (for example, a USB cable) can be monitored in a serial terminal window. Serial Monitor in Arduino IDE serves as a terminal, and with minimal code, this project was able to configure it to both monitor and control the system. This creates a unified interface, where both code and terminal are integrated into a single application, eliminating the need to constantly switch between programs. This significantly enhances the user experience.

4.4.3 Hardware control pin connections

Monarch 2 has four different possible UART hardware pin connection options, derived from possible combinations including/not including low-power and hardware flow control capabilities. For this project, it was desired to have both low-power and hardware flow control features. Therefore Type 1 configuration was chosen; read more about it in Monarch 2's "System Integration Guide" (B.5 page 1-4).

4.4.4 UART library

General communication and execution of AT commands between ESP32 and Monarch 2 is done over UART lines. The ESP32 microcontroller needs to be programmed to be able to utilise this data transfer method. Espressif Systems, the manufacturer of ESP32, provides an official IoT development framework known as ESP-IDF [82]. This framework includes the ESP-IDF UART library [83], and the ESP-AT library used for AT commands [84]. The initial appeal of this UART library was its detailed hardware flow control options.

However, both of these present the same issue related to the connection between ESP32 and Monarch 2.

The core problem is that this library treats the ESP32 as a Wi-Fi and Bluetooth modem and not as a host microcontroller, necessitating an additional microcontroller to act as the host. In this project, the ESP32 is the master microcontroller that controls the Monarch 2 modem. Specifically, the ESP32 acts as data Terminating Equipment (DTE), while the modem functions as the Data Communication Equipment (DCE). This means that the ESP32 role in the ESP-IDF UART (and subsequently ESP-AT) library and the actual projects are reversed, and the framework cannot be used. Additionally, this firmware is quite a large package, which is not ideal for students to need to download. Furthermore, it is wise to minimise the reliance on many different libraries during the development stages, as this makes it easier for Company of Things to expand on the project in the future.

The ESP32 add-on for Arduino IDE must be downloaded regardless. This add-on includes the HardwareSerial library [85]. Although it may appear simpler compared to ESP-IDF, this library allows for the required configuration of hardware flow control and other required specifications. Therefore, it was chosen for this project implementation. Serial flow control is often used with the ESP32, but the Monarch 2 documentation states that this would overwrite the hardware flow control, making this not an option (B.5). However, it is important to note that the choice of UART library is solely a software decision and does not impact the hardware connections. If a more suitable library is discovered or developed in the future, the HardwareSerial library can be substituted.

4.4.5 Design of webpage

The webpage code used for the Wi-Fi demonstration is largely based on a project from Random Nerd Tutorials [81]. This decision was made due to the fact that the construction of a webpage from scratch with HTML code is not the primary objective of this project. Another reason is that the project developer does not have experience in this field and would use an unnecessary amount of time to do it. The code is inspired by the tutorial but has been modified to better align with this project's requirements.

4.4.6 Creating own library for Monarch 2

A custom library for Monarch 2 was created for this project. The motivation behind this was to have more clear and manageable communication between ESP32 and Monarch 2, which is quite specific to this project. The approach of creating the library offered greater control and flexibility during the project development and allowed for easier expansion in the future.

4.4.7 Power source

Although it is possible to connect an external battery circuit to the ESP32 and Monarch 2, for the sake of simplified project development, they are powered through USB cables connected to a PC. The ESP32 utilises the same cable for both power supply and serial communication between itself and the PC.

4.4.8 ESP32 sleep mode

As explained in Chapter "4.2.4 ESP32 low-power", the ESP32 has different power saving modes. Deep sleep saves the most amount of power but has a big drawback. When the microcontroller enters this mode, all configurations from the setup section of the code are lost, including all AT command executions. This means that the setup code would need to be re-executed upon each wake-up, something that would take time. This does not align with the system requirements, as action should be carried out right after the system has been woken up by button presses. The light sleep mode, however, keeps the setup configurations, eliminating this problem. However, it consumes more power than deep sleep. This is a necessary compromise.

4.4.9 Why Monarch 2 mostly uses PSM and not eDRX

The main differences between Power Saving Mode (PSM) and extended Discontinuous Reception (eDRX) are how often the device is woken up and how deep it sleeps. eDRX listens for data more frequently, typically ranging from seconds to minutes. This also makes the device consume more power. PSM typically listens between minutes and hours. The demonstration setup for CoT DevKit IoT in this project only needs to be woken up and send data when the buttons are pushed. That is why it made more sense to test the system using PSM. Predefined functions for both PSM and eDRX are implemented in the CotT_Monarch library, enabling students to choose what fits their project best.

4.4.10 Why could not test with IoT-platform Deploii

One of the main objectives for this project was to establish connection and exchange data with Company of Thing's IoT platform Deploii. However, the platform is still under development and was not ready for NB-IoT communication at the time of testing. Therefore, this objective/work package could not be done and was replaced with the PING test. It is important to note that Deploii was able to work with Wi-Fi. However, testing the platform with an ESP32 was something Company of Things had done extensively and was therefore not necessary in this thesis.

4.4.11 Why all software is Arduino C

Arduino C is a "simplified" version of C++ designed to make it more approachable for beginners. An argument can be made that the core software functionalities developed throughout this project could have been written in more low-level embedded C code. This still allows the user to program their project in Arduino C on top of this. However, it is deemed more important to maintain a more open-source mentality. Fostering transparency for beginners and not just more advanced students. This also makes the development board even more adaptable and allows the students to gain an even deeper understanding of how it actually works.

4.4.12 Why evaluation boards instead of creating PCB from the beginning

A development board is essentially a printed circuit board (PCB). It is more efficient and practical to start the design process with evaluation kits and breadboards. PCB design can take a lot of time. When selecting different components and wire connections, it is easier to do this with a breadboard setup. Furthermore, the first iterations of PCB layouts often have mistakes. When troubleshooting, other mistakes can be falsely interpreted that the fault lies with the component connections, while in reality, it can, for example, be a missed ground connection. The approach of using evaluation kits and breadboards saves time, makes troubleshooting easier, and allows for a smoother transition into the actual PCB design.

5 Implementation and testing

This chapter encompasses the process of actually constructing the system with all challenges that occurred. The setup was incrementally implemented and sequentially tested. Thenceforth, combining all parts and testing the final setup with the demonstration system.

5.1 Process of implementation and testing throughout

Different steps were taken to arrive at the final CoT DevKit IoT proof-of-concept with a demonstration setup. This subchapter describes these steps and what occurred at each stage.

5.1.1 Initial test of Monarch 2

The first step involved conducting a direct setup and assessment of the NB-IoT modem, as illustrated in Figure 41. When connecting the evaluation kit to a PC for the first time, the requisite drivers were automatically installed. The Monarch 2 module's UART0 interface was linked to the PC via a USB cable. UART0 is the UART port on Monarch 2 used for AT commands. Unfortunately, the SIM cards dispatched by Telenor were misplaced in transit. However, the evaluation kit was accompanied by a global SIM card that could be employed for the preliminary testing phase. Subsequently, the awaited Telenor SIM card eventually arrived, facilitating subsequent testing procedures.

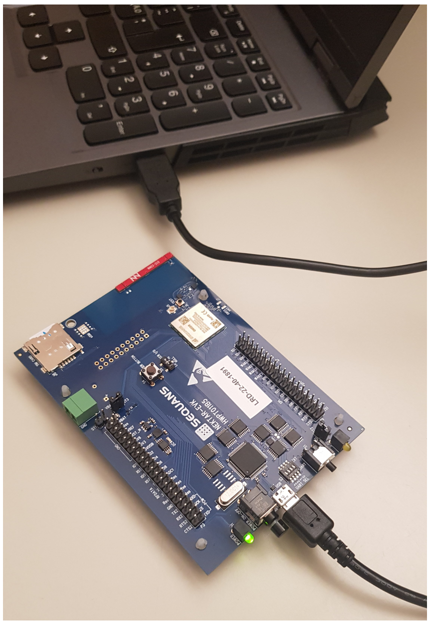
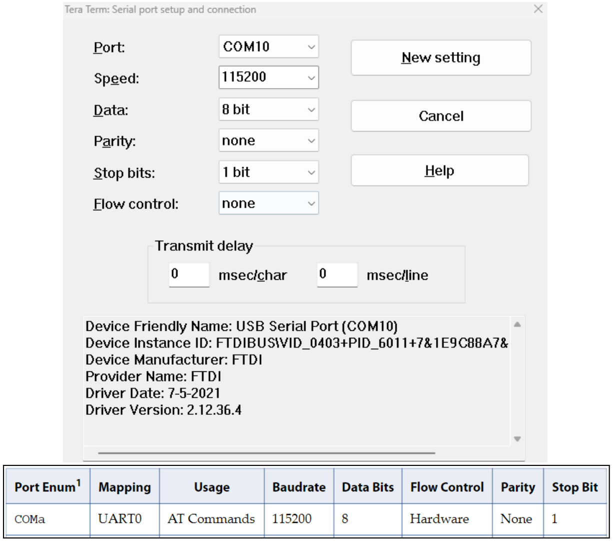


Figure 41: Setup

Controlling and configuring Monarch 2 was done with the utilisation of AT commands, which can be manually typed into a computer terminal. At this stage, the Tera Term serial terminal emulator program was employed [86]. Upon launching the program, the appropriate COM port for the USB cable was selected, and the necessary configuration settings for UART0 were adjusted under Setup > Serial port, see Figure 42. To see entered commands, the local echo functionality was activated under Setup > Terminal. See Figure 43.


Tera Term: Serial port setup and connection

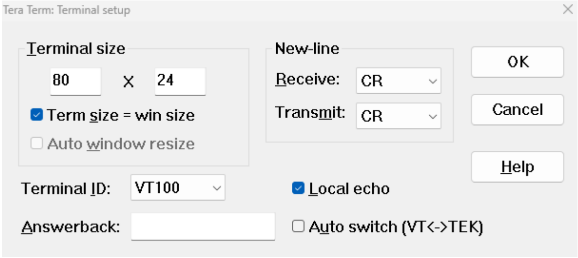
Port: COM10
Speed: 115200
Data: 8 bit
Parity: none
Stop bits: 1 bit
Flow control: none

Transmit delay
0 msec/char 0 msec/line

Device Friendly Name: USB Serial Port (COM10)
Device Instance ID: FTDIBUS\VID_0403+PID_6011+7&1E9C88A7&
Device Manufacturer: FTDI
Provider Name: FTDI
Driver Date: 7-5-2021
Driver Version: 2.12.36.4

Port Enum[1]	Mapping	Usage	Baudrate	Data Bits	Flow Control	Parity	Stop Bit
COMa	UART0	AT Commands	115200	8	Hardware	None	1

Figure 42: UART0 settings from datasheet implemented in Tera Term (B.3 page 4)


Tera Term: Terminal setup

Terminal size
80 X 24
☑ Term size = win size
□ Auto window resize

New-line
Receive: CR
Transmit: CR

Terminal ID: VT100
Answerback:

☑ Local echo
□ Auto switch (VT<->TEK)

Figure 43: Local echo enabled

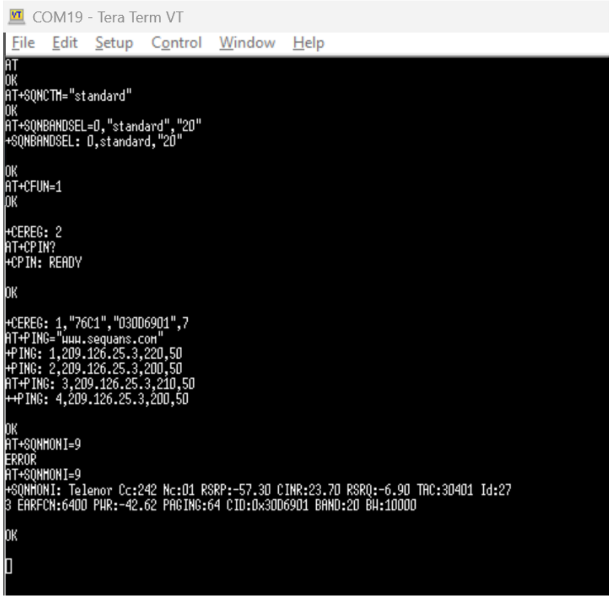
In order to verify the operational status of the module, several functionality tests AT commands were transmitted as shown in Figure 49. Table 10 briefly explains the commands. For more details, see the "Monarch 2 AT Command Use Cases" document in Attachment B.4. Encouragingly, the modem performed as expected, exhibiting the desired behavior and functionality. However, the narrowband network was not always stable.

AT command:	Meaning:
AT	See if computer (serial port) and module are connected properly
AT+SQNCTM	Select operation mode
AT+SQNBANDSEL	Select appropriate bands
AT+CPIN?	Check if SIM is inserted and unlocked
AT+CFUN=1	Attach to network
AT+PING	Check network connection
AT+SQNMONI	Network scan

Table 10: Meaning of functionality test AT commands (see B.4)

5.1.2 Control Monarch 2 with ESP32

The next step involved controlling Monarch 2 with ESP32, using Serial Monitor in Arduino IDE as a terminal that works with the system. This communication between Monarch 2



```
AT
OK
AT+SQNCTM="standard"
OK
AT+SQNBANDSEL=0,"standard","20"
+SQNBANDSEL: 0,standard,"20"

OK
AT+CFUN=1
OK

+CEREG: 2
AT+CPIN?
+CPIN: READY

OK

+CEREG: 1,"76C1","030D6901",7
AT+PING="www.sequans.com"
+PING: 1,209.126.25.3,220,50
+PING: 2,209.126.25.3,200,50
AT+PING: 3,209.126.25.3,210,50
++PING: 4,209.126.25.3,200,50

OK
AT+SQNMONI=9
ERROR
AT+SQNMONI=9
+SQNMONI: Telenor Cc:242 Nc:01 RSRP:-57.30 CINR:23.70 RSRQ:-6.90 TAC:30401 Id:27
3 EARFCN:6400 PWR:-42.62 PAGING:64 CID:0x30D6901 BAND:20 BW:10000

OK
```

Figure 44: Functionality test with AT commands

and ESP32 is the core functionality of the entire project in this thesis. Everything related to NB-IoT is built upon it. This entailed establishing physical UART connections between the two components, configuring UART communication code to control Monarch 2 with ESP32, and creating some code to modify the Serial Monitor in Arduino IDE to provide similar functionalities as Tera Term (but connected to ESP32 and not Monarch 2). This enabled the input of AT commands and reading of responses in the Serial Monitor. This is also to confirm that ESP32 and Monarch 2 can operate together.

The pin connections can be found in Chapter "4.2 Detailed system design: Hardware, Figure 21. Note that only the UART pins from the figure are attached at this stage, and not the different buttons. The physical setup is shown in Figure 45 and 46. In this case, the communication with Monarch 2 bypasses the USB cable and instead utilises the physical pins. To enable this configuration, the UART0 switch "S3" on the evaluation kit must be set to "EXT," as depicted in Figure 46. The power source is still the PC through the USB cable.

The code created to control Monarch 2 with ESP32 through Serial Monitor was not long, and the most important parts are shown in the figures below in Figure 47 and 48. This code file, called "type_at_esp32_monarch2.ino", is attached to the submission of this thesis. See Attachment B.10. The UART library "HardwareSerial" from Arduino was utilised. Different libraries, like Espressif IoT Development Framework designed for ESP32, were tested but could not get them to work. Read more about this decision in Chapter "4.4.4 UART library".

The code in Figure 47 defines, with class "HardwareSerial", the different UART ports between ESP32 and, respectively, Monarch 2 and the PC. Figure 48 encompasses the "void loop()". Here the UART line between PC and ESP32, "esp32Serial", is constantly

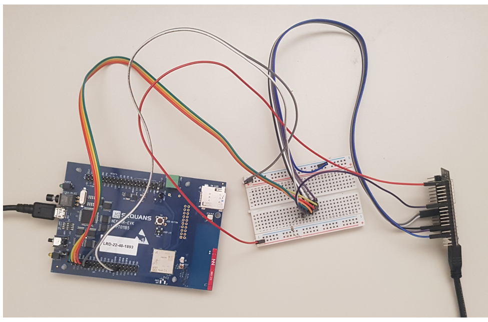


Figure 45: UART setup

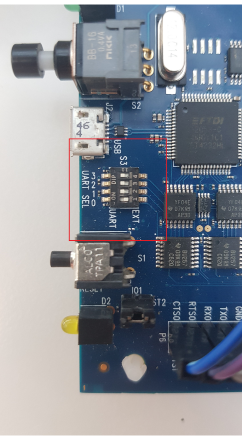


Figure 46: UART switch

monitored to see if anything was entered in Serial Monitor. If an AT command is typed, it is forwarded to Monarch 2. It also echoes the command so that users can see what they typed. The UART line between Monarch 2 and ESP32, "monarchSerial", is also constantly monitored, listening for data from Monarch 2. If anything comes on this line, like a response for the AT command, it is displayed in Serial Monitor.

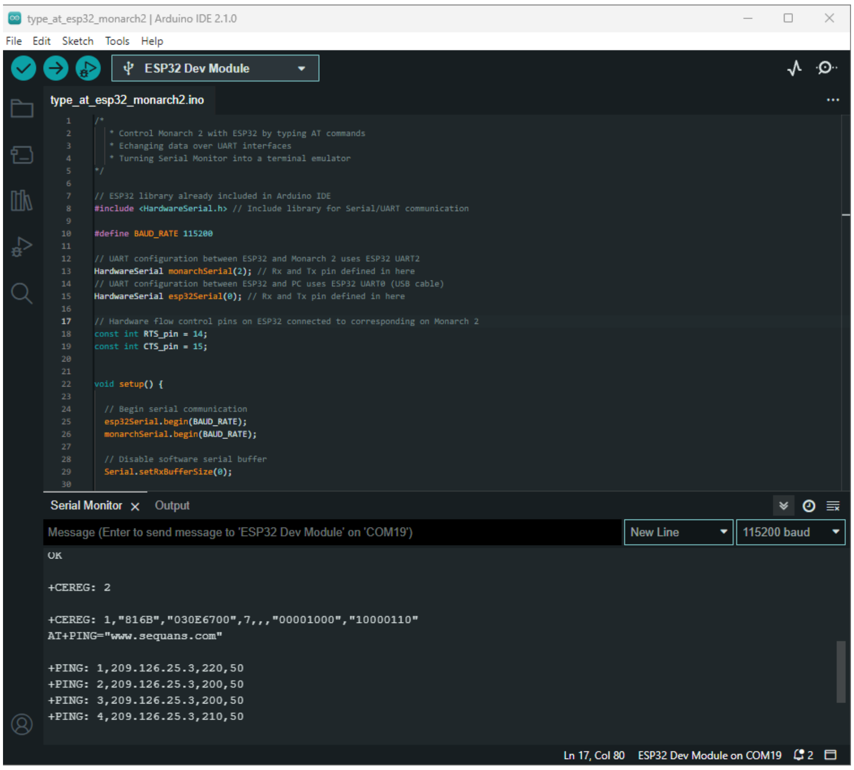
```
// UART configuration between ESP32 and Monarch 2 uses ESP32 UART2
HardwareSerial monarchSerial(2);
// UART configuration between ESP32 and PC uses ESP32 UART0 (USB cable)
HardwareSerial esp32Serial(0);
```

Figure 47: UART port selection

```
void loop() {

  // Forward from ESP32 to Monarch 2
  // Type AT command in Serial Monitor
  if (esp32Serial.available()) {
    char c = esp32Serial.read();
    monarchSerial.write(c); // Forward data
    esp32Serial.write(c); // Echo
  }

  // Forward data from Monarch 2 to ESP32
  // Read AT response in Serial Monitor
  if (monarchSerial.available()) {
    char c = monarchSerial.read();
    esp32Serial.write(c); // Forward data
  }
}
```

Figure 48: Type AT commands in Serial Monitor

With this program, the UART communication lines are established, but not the modem configurations. Monarch 2 needs to be configured with AT commands that initiate UART, configure operation mode, and connect to the network (see Chapter 4.3.2). When all these commands are executed, the system can actually operate. The network scan and connection test from the last step was carried out successfully, as shown in Figure 49.


type_at_esp32_monarch2 | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
type_at_esp32_monarch2.ino
Serial Monitor
Output
New Line
115200 baud
Ln 17, Col 80
ESP32 Dev Module on COM19
2

```
/*
  * Control Monarch 2 with ESP32 by typing AT commands
  * Echanging data over UART interfaces
  * Turning Serial Monitor into a terminal emulator
*/

// ESP32 library already included in Arduino IDE
#include <HardwareSerial.h> // Include library for Serial/UART communication

#define BAUD_RATE 115200

// UART configuration between ESP32 and Monarch 2 uses ESP32 UART2
HardwareSerial monarchSerial(2); // Rx and Tx pin defined in here
// UART configuration between ESP32 and PC uses ESP32 UART0 (USB cable)
HardwareSerial esp32Serial(0); // Rx and Tx pin defined in here

// Hardware flow control pins on ESP32 connected to corresponding on Monarch 2
const int RTS_pin = 14;
const int CTS_pin = 15;


void setup() {

  // Begin serial communication
  esp32Serial.begin(BAUD_RATE);
  monarchSerial.begin(BAUD_RATE);

  // Disable software serial buffer
  Serial.setRxBufferSize(0);

```

Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')

```
OK

+CEREG: 2

+CEREG: 1,"816B","030E6700",7,,,"00001000","10000110"
AT+PING="www.sequans.com"

+PING: 1,209.126.25.3,220,50
+PING: 2,209.126.25.3,200,50
+PING: 3,209.126.25.3,200,50
+PING: 4,209.126.25.3,210,50
```

Figure 49: Response Serial Monitor

5.1.3 NB-IoT button

In addition to the existing code, a fundamental NB-IoT button was added to the project. Please refer to Figure 21 for the button's connection and physical setup in Figure 50.

The purpose of this button is to test the ESP32 and Monarch 2 system's ability to connect and transmit data over the narrowband network. This was originally going to be done with the IoT platform, Deploii, but it was not ready for an NB-IoT connection at the time of testing. Therefore, it was replaced with a PING test to a website URL (Uniform Resource Locator).

The "handleResponse()" function was created to monitor the UART lines and print responses from Monarch 2 in Serial Monitor. This also included incoming ULC (Uplink Commands). The first iteration of the NB-IoT button code snippet did not have proper flow control logic but was added in the subsequent iteration. This required its own response function called "buttonNB_handleResponse_sendPING()". The differences in complexity can be seen by comparing Figure 51 below and Figure 38 in Chapter 4.3.4. The final NB-IoT button code snippet also verified if the modem had woken up.

During this stage, the Telenor SIM cards that Company of Things had bought arrived. However, there were some connectivity issues with the SIM cards intermittently working for certain weeks and not working at other times. In an attempt to resolve the issue, a Telenor SIM card borrowed from supervisor Nils Kristian Rossing was also utilised - switching between the different cards. The narrowband IoT network was stable when testing Monarch 2 alone.

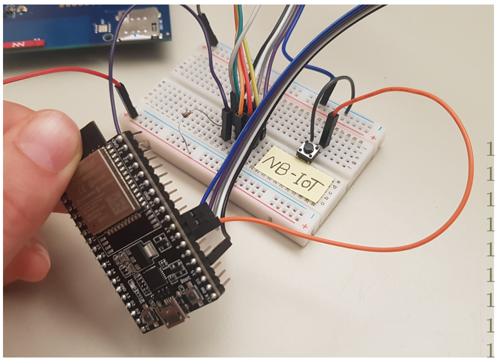


Figure 50: NB-IoT button setup

```
// Read current button state
buttonState = digitalRead(BUTTON_pin);

// If button pressed, send PING test
if (buttonState != lastButtonState && buttonState == LOW) {
  Serial.println("NB-IoT␣button␣pushed");

  // Send command to the modem
  String pingCommand = "AT+PING=\"www.sequans.com\"";
  monarchSerial.println(pingCommand);

  // Echo the command
  Serial.println("Command:␣");
  Serial.println(pingCommand);

  handleResponse();
  delay(100);
}

lastButtonState = buttonState; // Update button state
```

Figure 51: First iteration NB-IoT button code (see Figure 38 for final version)

5.1.4 Creating AT command functions and CoT_Monarch library

Typing in AT commands manually can get tiresome, as many Monarch 2 configurations require many different commands. This is especially relevant when powering up the system. Therefore, a set of predefined functions with different configuration AT commands were created. These functions are described in Chapter 4.3.2. This allows users to simply only type in the function name in Serial Monitor instead of each AT command individually. Single commands were still possible to type in.

Significant modification to the original code, shown in Figure 48, was necessary to implement this new feature. The main change was being able to distinguish between AT commands and function names. This was done by examining the first two characters of the input, as all AT commands begin with "AT". The original code used "Serial.write()", which sends raw bytes, one at a time. This function allows for more control but cannot handle strings. The modified code, therefore, switched to "Serial.print()" which sends an ASCII-encoded version of the string. The data flow control challenge was improved by employing trimming techniques and more precise timing.

The demo with only the NB-IoT button is attached to this submission. See Attachment B.11. All of these AT functions were stored in the CoT_Monarch library files for the final system.

5.1.5 Wi-Fi test

The Wi-Fi element of the project works without Monarch 2. Therefore it was subsequently tested without it first. Testing that the ESP32 could find the local network, the built-in "WiFiScan" example was executed. It is found in Arduino IDE under File > Examples > WiFi > WifiScan. The hardware setup was only the ESP32 connected to the PC. See Figure 52.

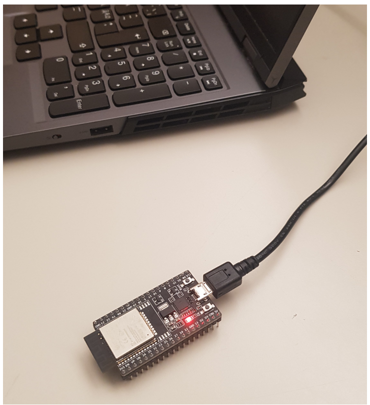


Figure 52: Wi-Fi test setup

5.1.6 Wi-Fi button

Detailed information on setting up the ESP32 as a web server that hosts a webpage can be found in Chapter 4.3.3. For the code snippet related to the Wi-Fi button, refer to Chapter 4.3.4. The demonstration code specifically designed for the Wi-Fi button, excluding other functionalities, is attached to this thesis submission. See Attachment B.12.

The first version of the code is similar to the final. When the button is pressed, the webpage button state is updated accordingly. Observant readers may notice that the final code still includes a test point named "wifi_output_point", which is the same variable used for the webpage button. This pin, GPIO 2 on the ESP32, was also connected to a LED (Light Emitting Diode) to verify the changes, which turned the LED on and off. See Figure 53. Additionally, users could also click the button on the wabpage and send a signal back and see the LED change. Everything was also printed out in Serial Monitor.

When the Wi-Fi button was operational, it was combined with the NB button and AT command/function type code. As well as placing the buttons on the same breadboard.

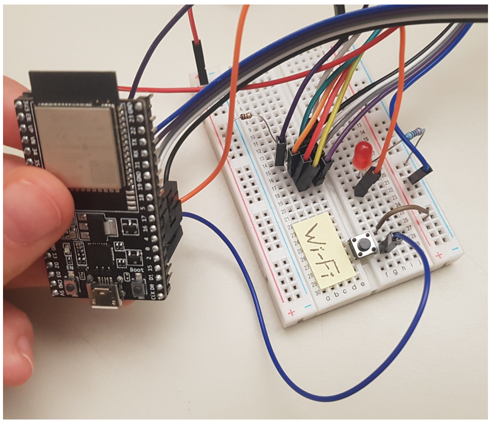


Figure 53: Wi-Fi button setup

5.1.7 Monarch 2 low-power

The implementation of power-saving techniques for Monarch 2 was the next step. Specifically, Power Saving Mode (PSM) and extended Discontinuous Reception (eDRX) were incorporated into the system. While both functionalities were made available to the user, PSM was predominantly used due to specific reasons outlined in Chapter 4.4.9.

Two separate AT command functions, one for PSM named "initPSM" and the other for eDRX named "init_eDRX", were developed and added to the CoT_Monarch library. The PSM function was placed in the setup section of the code, enabling Monarch 2 to enter sleep mode right after startup configurations were complete. Additionally, the logic for hardware flow control pins was implemented into the NB-IoT button, as demonstrated in the NB button code presented in Chapter 4.3.4, Figure 38. The module was woken up by toggling the RT pin, which was configured as the wake-up source. The changes in power saving states were verified by monitoring the RTS and CTS values in Serial Monitor.

Furthermore, an LED was connected to the "PS_status" pin on Monarch 2 to indicate when the module was in active mode, as shown in Figure 68. However, it should be noted that 1,8V was emitted from this line, which was the theoretical minimum voltage required to power the LED.

5.1.8 ESP32 low-power

The last button added to the system was the "ESP wake up" button, as shown in Figure 54. The goal was to not only conserve energy in the Monarch 2 module but also with the ESP32 and, consequently, the entire system. The Wi-Fi and NB-IoT buttons were originally designed to simulate sensor data. Therefore, when there was no data being transmitted, indicated by neither the Wi-Fi nor the NB-IoT button being pressed within a specific timeframe, the system would enter a sleep mode to conserve power.

Further details regarding the code implementation and its functionality can be found in Chapter 4.3.4. This feature is an integral part of the final system and will be further elaborated upon in the next Chapter "5.2 Testing and validation of final Demonstration System".

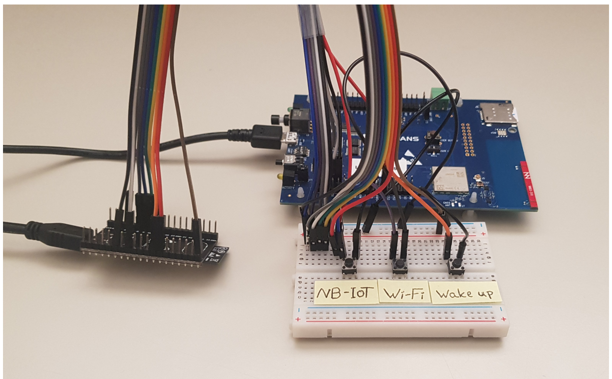


Figure 54: Adding ESP32 wake up button

5.2 Testing and validation of final Demonstration System

Implementing the system consisted of an iterative design and systematic approach. The previous Subchapter explained the process of building the system. For each step, there were tests to verify the functionality. This Subchapter will look at how the final system, meaning the CoT DevKit IoT proof-of-concept with demonstration setup, was tested. Please refer to Chapter "4.1.3 Finalised setup" for physical setup in Figure 18, webpage in Figure 19, and Figure 20 for Arduino IDE with Serial Monitor interface. The final code is delivered with this thesis submission, see Attachment B.12, and is also printed out in Attachment B.13 for readers that do not have access to the separate files.

The final system was within itself a test setup for development board proof-of-concept, with the objective of showing that ESP32 and Monarch 2 could be integrated into a board that has all the system requirements: Wi-Fi and narrowband capabilities, being able to program in Arduino C and low power modes to save energy. The UART with hardware flow control line between ESP32 and Monarch 2 showed that the modules were actually compatible and could communicate. The Wi-Fi and NB-IoT buttons were examples of both network connectivity, mirroring the functionality of sensors that students can incorporate into their projects. The ESP32 wake-up button and Monarch 2's power-saving functions reveal the low-power capabilities. Testing the final system consisted of just doing the different things it was supposed to do. Verifying the functionalities with what happened with the buttons, responses in Serial Monitor, changes on the webpage, and some physical LEDs.

5.2.1 ESP32 and Monarch 2 compatibility

The initial tests were conducted to verify the compatibility of the ESP32 and Monarch 2 components. The ESP32 acted as the DTE (Data Termination Equipment) and served as the master of the system and controlled Monarch 2. Monarch 2 was the DCE (Data Communication Equipment). To perform these tests, AT commands and function names were manually entered into the Serial Monitor, and the corresponding responses were observed. These tests also indirectly tested if Arduino C was a good program to write the code in and able to utilise Serial Monitor as a terminal window. This testing stage was a crucial part of the entire system.

5.2.2 NB-IoT functionality

Some of the relevant testing with the narrowband network was conducted in the previous Subchapter, "5.2.1 ESP32 and Monarch 2 compatibility".

Moving forward, the testing focused on the NB-IoT button by pressing it and verifying whether it successfully transmitted a connectivity test signal (PING) to the website "www.sequans.com." This test aimed to demonstrate that the system was capable of sending data from the ESP32 to the Monarch 2 module beyond just sending AT commands. This was done both inside at Realfagsbygget Gløshaugen NTNU and outside at Tømmerdalen in Trondheim, as shown in Figure 55. Tømmerdalen was specifically chosen because it is a weak spot in the Telenor network; see Figure 56. Figure 57 shows the setup outside, inside was the same as before.

When the button was pushed, feedback was displayed in Serial Monitor. Including if Monarch 2 woke up, sent a PING test, and subsequently went back to sleep. Furthermore, the toggling of RTS and CTS pins was also printed out to verify this. RTS was toggled to wake up, CTS checked if the module was ready for data exchange, and a response from "AT-CFUN?" checked the network connection status.

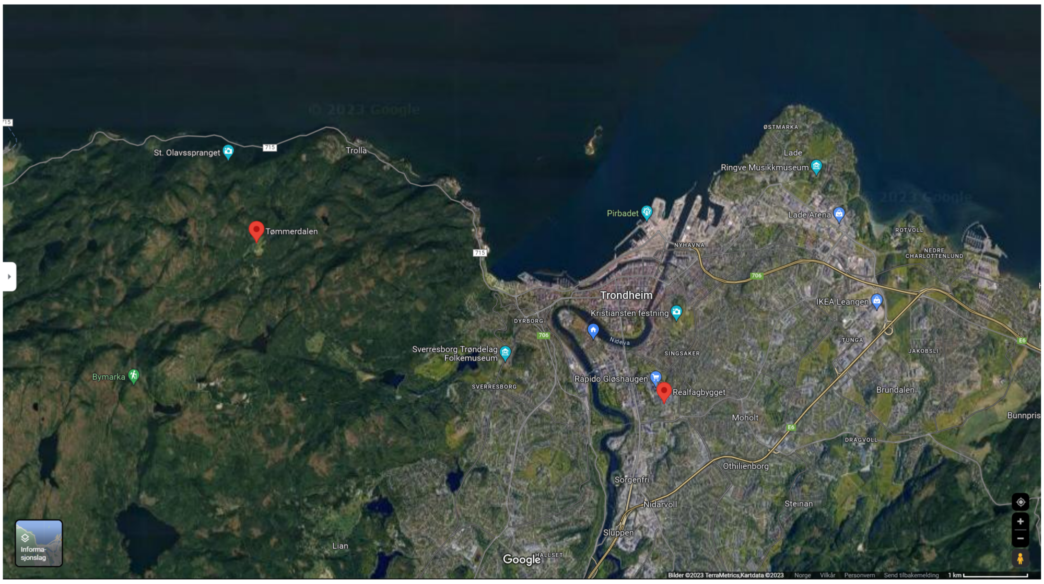


Figure 55: Tømmerdalen and Realfagsbygget on Google Maps

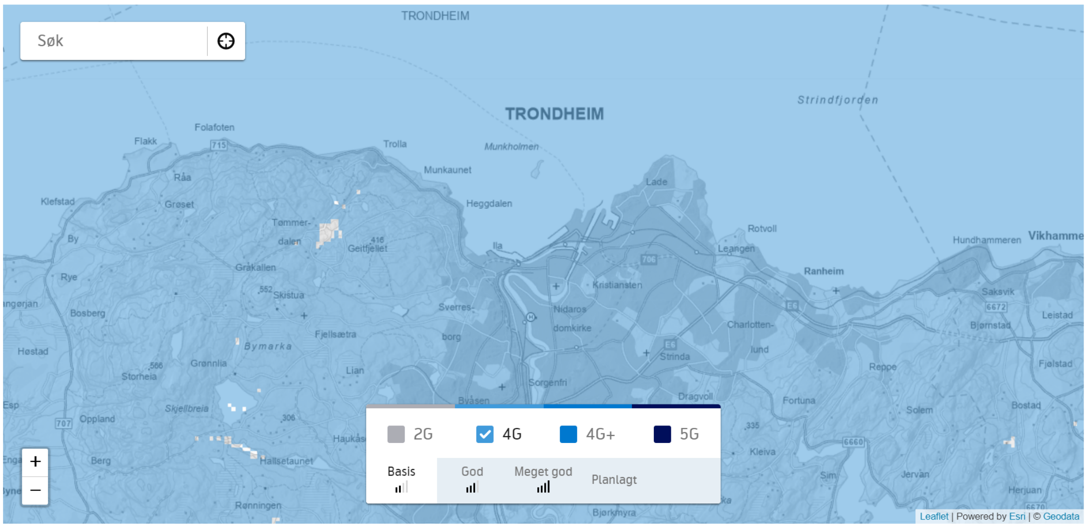


Figure 56: Telenor coverage map of Trondheim [27]

Figure 57: Setup outside in Tømmerdalen

5.2.3 Monarch 2 power saving

Some of the relevant tests overlapped with the previous Subchapter, "5.2.2 NB-IoT functionality", specifically regarding the observation of the Monarch module being woken up when the NB-IoT button was pressed and subsequently going back to sleep.

When the system was powered up, the setup section of the code was executed. The setup section included the initialisation function for Power Saving Mode, which aimed at putting Monarch 2 to sleep right after the setup configurations were done. AT commands were included in the function, and their responses were printed out in Serial Monitor. This allowed for constant monitoring and verification.

A LED was connected to the PS_status pin on Monarch 2 to see how it illuminated depending on Monarch 2's power state. This pin delivers a maximum of 1,8V (B.2 page 10). This should theoretically just work, but a more extended cable connection was replaced with directly holding it against the pin to eliminate as much resistance as possible.

5.2.4 Wi-Fi functionality

Testing the Wi-Fi functionality involved verifying whether the ESP32 could successfully connect to the local network, act as a web server, host a webpage, and ensure that the state of the physical button corresponded to the state displayed on the webpage. This was done in Ralfagsbygget at Gløshaugen NTNU.

The ESP32 should establish a connection to the network and print its IP (Internet Protocol) address in Serial Monitor during the first execution of the code's main loop. The user then copies this IP and pastes it into a web browser, as this is the webpage's URL.

5.2.5 ESP32 power saving

The ESP32 was programmed to enter light sleep mode after one minute of inactivity, where none neither the NB-IoT nor Wi-Fi buttons were pushed. The testing procedure consisted of waiting one minute and observing activity in Serial Monitor. After that, pressing the NB-IoT and Wi-Fi button and reloading the webpage to see if they did not work to confirm that the system had gone into deep sleep. Lastly, the ESP32 wake-up button was pressed, and the subsequent same tests were done to see if the system had been woken up.

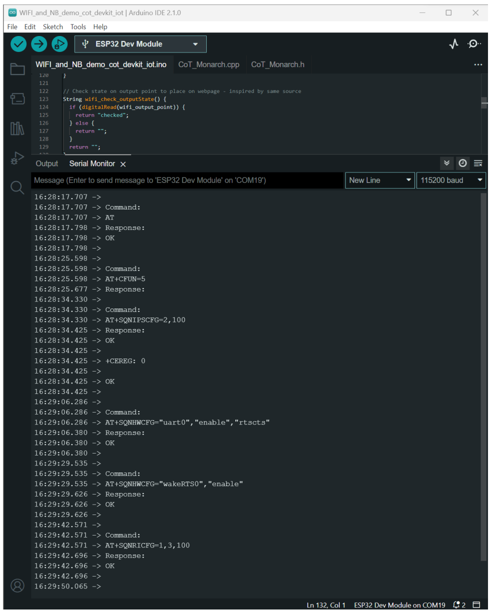
6 Results

Results and data are acquired from the tests conducted in Chapter "5.2 Testing and validation of final Demonstration System." The figures below in this chapter depict images of the Arduino IDE with the Serial Monitor, while a full printout of the Serial Monitor can be found in the appendices.

6.1 ESP32 and Monarch 2 compatibility

AT command executions are done to demonstrate ESP32 and Monarch 2 compatibility since this represents if they are able to communicate with each other.

Manually enter AT commands in Serial Monitor
Figure 58 showcases the last manually entered AT commands in the Serial Monitor, along with the corresponding responses, which were used to configure the UART functionalities on the Monarch 2 module. See Appendix A.14 for a full printout.



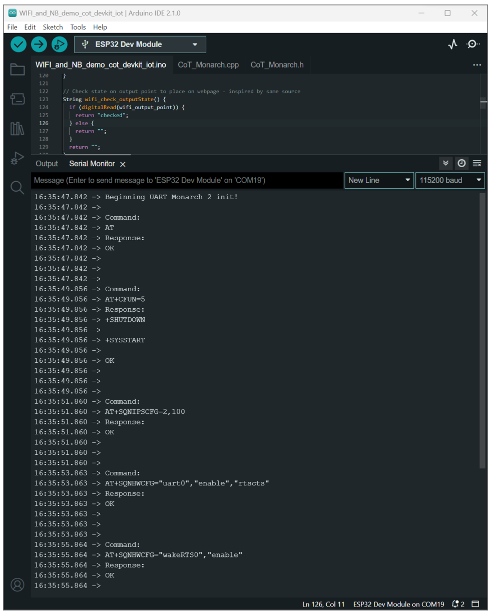
WIFI_and_NB_demo_cot_devkit_iot | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 132, Col 1
ESP32 Dev Module on COM19

```
}

// Check state on output point to place on webpage - inspired by same source
String wifi_check_outputState() {
  if (digitalRead(wifi_output_point)) {
    return "checked";
  } else {
    return "";
  }
  return "";
```

```
16:28:17.707 ->
16:28:17.707 -> Command:
16:28:17.707 -> AT
16:28:17.798 -> Response:
16:28:17.798 -> OK
16:28:17.798 ->
16:28:25.598 ->
16:28:25.598 -> Command:
16:28:25.598 -> AT+CFUN=5
16:28:25.677 -> Response:
16:28:34.330 ->
16:28:34.330 -> Command:
16:28:34.330 -> AT+SQNIPSCFG=2,100
16:28:34.425 -> Response:
16:28:34.425 -> OK
16:28:34.425 ->
16:28:34.425 -> +CEREG: 0
16:28:34.425 ->
16:28:34.425 -> OK
16:28:34.425 ->
16:29:06.286 ->
16:29:06.286 -> Command:
16:29:06.286 -> AT+SQNHWCFG="uart0","enable","rtscts"
16:29:06.380 -> Response:
16:29:06.380 -> OK
16:29:06.380 ->
16:29:29.535 ->
16:29:29.535 -> Command:
16:29:29.535 -> AT+SQNHWCFG="wakeRTS0","enable"
16:29:29.626 -> Response:
16:29:29.626 -> OK
16:29:29.626 ->
16:29:42.571 ->
16:29:42.571 -> Command:
16:29:42.571 -> AT+SQNRICFG=1,3,100
16:29:42.696 -> Response:
16:29:42.696 -> OK
16:29:42.696 ->
16:29:50.065 ->
```

Figure 58: Type AT commands manually

Manually enter function name in Serial Monitor
Figure 60 illustrates the same set of last commands, but this time they were executed by typing the "uartInitMonarch" function name. See Appendix A.13 for full print out. This was also done with the other predefined functions.


WIFI_and_NB_demo_cot_devkit_iot | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 126, Col 11
ESP32 Dev Module on COM19

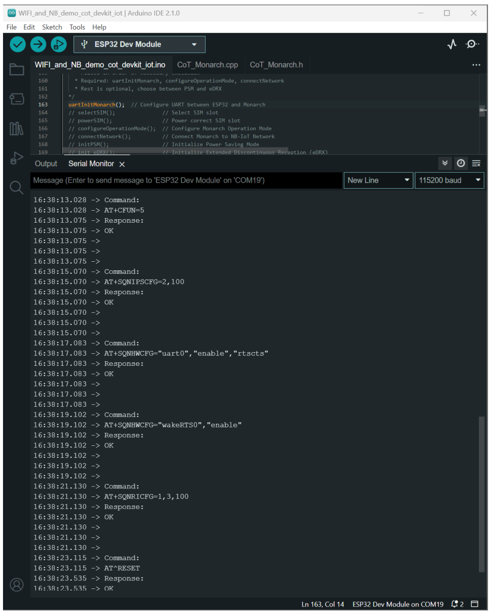
```
}

// Check state on output point to place on webpage - inspired by same source
String wifi_check_outputState() {
  if (digitalRead(wifi_output_point)) {
    return "checked";
  } else {
    return "";
  }
  return "";
```

```
16:35:47.842 -> Beginning UART Monarch 2 init!
16:35:47.842 ->
16:35:47.842 -> Command:
16:35:47.842 -> AT
16:35:47.842 -> Response:
16:35:47.842 -> OK
16:35:47.842 ->
16:35:47.842 ->
16:35:47.842 ->
16:35:49.856 -> Command:
16:35:49.856 -> AT+CFUN=5
16:35:49.856 -> Response:
16:35:49.856 -> +SHUTDOWN
16:35:49.856 ->
16:35:49.856 -> +SYSSTART
16:35:49.856 ->
16:35:49.856 -> OK
16:35:49.856 ->
16:35:49.856 ->
16:35:49.856 ->
16:35:51.860 -> Command:
16:35:51.860 -> AT+SQNIPSCFG=2,100
16:35:51.860 -> Response:
16:35:51.860 -> OK
16:35:51.860 ->
16:35:51.860 ->
16:35:51.860 ->
16:35:53.863 -> Command:
16:35:53.863 -> AT+SQNHWCFG="uart0","enable","rtscts"
16:35:53.863 -> Response:
16:35:53.863 -> OK
16:35:53.863 ->
16:35:53.863 ->
16:35:53.863 ->
16:35:55.864 -> Command:
16:35:55.864 -> AT+SQNHWCFG="wakeRTS0","enable"
16:35:55.864 -> Response:
16:35:55.864 -> OK
16:35:55.864 ->
```

Figure 59: Type function name manually

Automatical function execution within code

Lastly, Figure 60 also presents the same set of AT commands, but in this case, the "uartInitMonarch" function was placed within the "void setup()" section of the code, being executed upon system power-up. The full printout can be found within Appendix A.11. This was also done with the other predefined functions.


WIFI_and_NB_demo_cot_devkit_iot | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 163, Col 14
ESP32 Dev Module on COM19

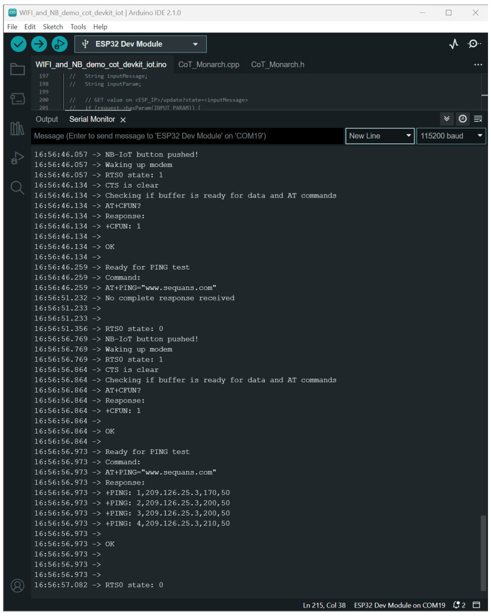
```
   * Required: uartInitMonarch, configureOperationMode, connectNetwork
   * Rest is optional, choose between PSM and eDRX
  */
  uartInitMonarch();  // Configure UART between ESP32 and Monarch
  // selectSIM();               // Select SIM slot
  // powerSIM();                // Power correct SIM slot
  // configureOperationMode();  // Configure Monarch Operation Mode
  // connectNetwork();          // Connect Monarch to NB-IoT Network
  // initPSM();                 // Initialize Power Saving Mode
  // init_eDRX();               // Initialize Extended Discontinuous Reception (eDRX)
```

```
16:38:13.028 -> Command:
16:38:13.028 -> AT+CFUN=5
16:38:13.075 -> Response:
16:38:13.075 -> OK
16:38:13.075 ->
16:38:13.075 ->
16:38:13.075 ->
16:38:15.070 -> Command:
16:38:15.070 -> AT+SQNIPSCFG=2,100
16:38:15.070 -> Response:
16:38:15.070 -> OK
16:38:15.070 ->
16:38:15.070 ->
16:38:15.070 ->
16:38:17.083 -> Command:
16:38:17.083 -> AT+SQNHWCFG="uart0","enable","rtscts"
16:38:17.083 -> Response:
16:38:17.083 -> OK
16:38:17.083 ->
16:38:17.083 ->
16:38:17.083 ->
16:38:19.102 -> Command:
16:38:19.102 -> AT+SQNHWCFG="wakeRTS0","enable"
16:38:19.102 -> Response:
16:38:19.102 -> OK
16:38:19.102 ->
16:38:19.102 ->
16:38:19.102 ->
16:38:21.130 -> Command:
16:38:21.130 -> AT+SQNRICFG=1,3,100
16:38:21.130 -> Response:
16:38:21.130 -> OK
16:38:21.130 ->
16:38:21.130 ->
16:38:21.130 ->
16:38:23.115 -> Command:
16:38:23.115 -> AT^RESET
16:38:23.535 -> Response:
16:38:23.535 -> OK
```

Figure 60: Function placed in code

6.2 NB-IoT functionality

Figure 61 displays the response observed in the Serial Monitor when the NB-IoT button was pressed inside Realfagsbygget at Gløshaugen. For a full printout, see Appendix A.12. Figure 62 depicts the Telenor coverage map over Tømmerdalen. The line and pins in Figure 63 is where the button was pressed outside in Tømmerdalen. Figure 64 shows the response on the purple pin, and Figure 65 depicts the response on the red pin.


WIFI_and_NB_demo_cot_devkit_iot | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 215, Col 38
ESP32 Dev Module on COM19
2

```
//   String inputMessage;
//   String inputParam;

//   // GET value on <ESP_IP>/update?state=<inputMessage>
//   if (request->hasParam(INPUT_PARAM)) {
```

```
16:56:46.057 -> NB-IoT button pushed!
16:56:46.057 -> Waking up modem
16:56:46.057 -> RTS0 state: 1
16:56:46.134 -> CTS is clear
16:56:46.134 -> Checking if buffer is ready for data and AT commands
16:56:46.134 -> AT+CFUN?
16:56:46.134 -> Response:
16:56:46.134 -> +CFUN: 1
16:56:46.134 ->
16:56:46.134 -> OK
16:56:46.134 ->
16:56:46.259 -> Ready for PING test
16:56:46.259 -> Command:
16:56:46.259 -> AT+PING="www.sequans.com"
16:56:51.232 -> No complete response received
16:56:51.233 ->
16:56:51.233 ->
16:56:51.356 -> RTS0 state: 0
16:56:56.769 -> NB-IoT button pushed!
16:56:56.769 -> Waking up modem
16:56:56.769 -> RTS0 state: 1
16:56:56.864 -> CTS is clear
16:56:56.864 -> Checking if buffer is ready for data and AT commands
16:56:56.864 -> AT+CFUN?
16:56:56.864 -> Response:
16:56:56.864 -> +CFUN: 1
16:56:56.864 ->
16:56:56.864 -> OK
16:56:56.864 ->
16:56:56.973 -> Ready for PING test
16:56:56.973 -> Command:
16:56:56.973 -> AT+PING="www.sequans.com"
16:56:56.973 -> Response:
16:56:56.973 -> +PING: 1,209.126.25.3,170,50
16:56:56.973 -> +PING: 2,209.126.25.3,200,50
16:56:56.973 -> +PING: 3,209.126.25.3,200,50
16:56:56.973 -> +PING: 4,209.126.25.3,210,50
16:56:56.973 ->
16:56:56.973 -> OK
16:56:56.973 ->
16:56:56.973 ->
16:56:56.973 ->
16:56:57.082 -> RTS0 state: 0
```

Figure 61: NB-IoT button pressed inside Realfagsbygget

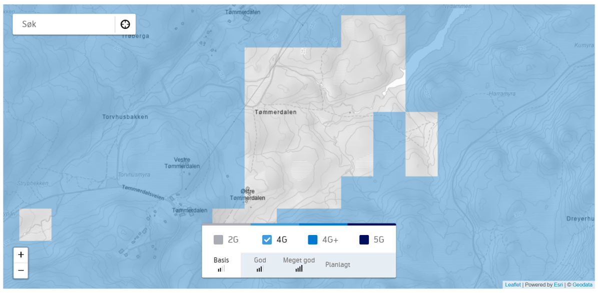


Figure 62: Telenor coverage map Tømmerdalen [27]

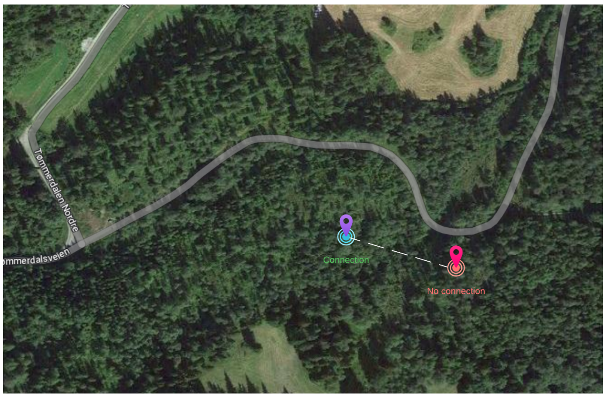
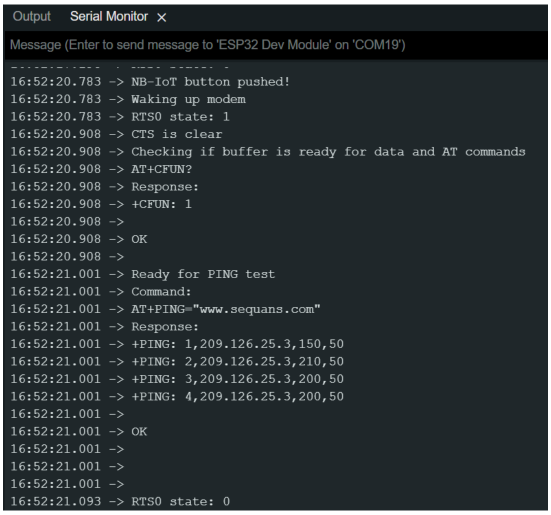


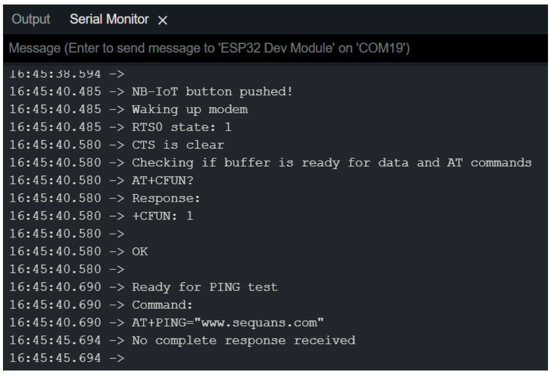
Figure 63: Google maps with pins



Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')

```
16:52:20.783 -> NB-IoT button pushed!
16:52:20.783 -> Waking up modem
16:52:20.783 -> RTS0 state: 1
16:52:20.908 -> CTS is clear
16:52:20.908 -> Checking if buffer is ready for data and AT commands
16:52:20.908 -> AT+CFUN?
16:52:20.908 -> Response:
16:52:20.908 -> +CFUN: 1
16:52:20.908 ->
16:52:20.908 -> OK
16:52:20.908 ->
16:52:21.001 -> Ready for PING test
16:52:21.001 -> Command:
16:52:21.001 -> AT+PING="www.sequans.com"
16:52:21.001 -> Response:
16:52:21.001 -> +PING: 1,209.126.25.3,150,50
16:52:21.001 -> +PING: 2,209.126.25.3,210,50
16:52:21.001 -> +PING: 3,209.126.25.3,200,50
16:52:21.001 -> +PING: 4,209.126.25.3,200,50
16:52:21.001 ->
16:52:21.001 -> OK
16:52:21.001 ->
16:52:21.001 ->
16:52:21.001 ->
16:52:21.093 -> RTS0 state: 0
```

Figure 64: Response at purple pin



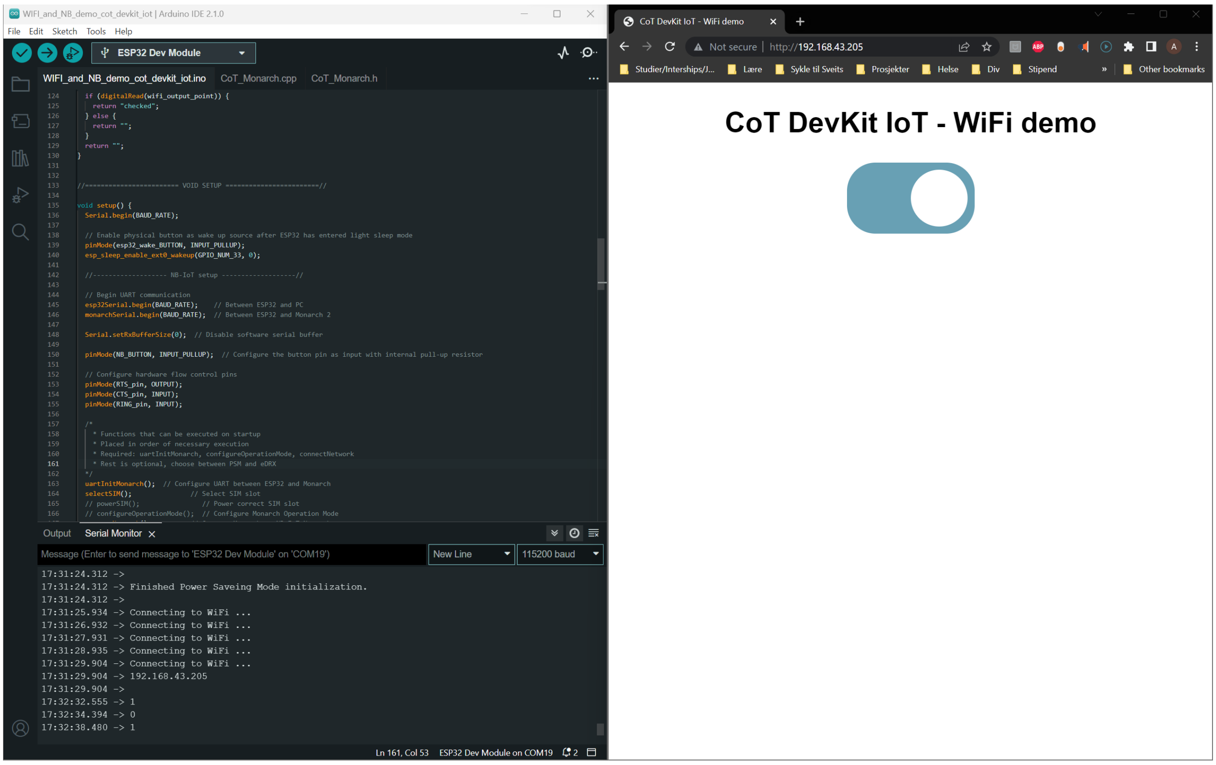
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')

```
16:45:38.594 ->
16:45:40.485 -> NB-IoT button pushed!
16:45:40.485 -> Waking up modem
16:45:40.485 -> RTS0 state: 1
16:45:40.580 -> CTS is clear
16:45:40.580 -> Checking if buffer is ready for data and AT commands
16:45:40.580 -> AT+CFUN?
16:45:40.580 -> Response:
16:45:40.580 -> +CFUN: 1
16:45:40.580 ->
16:45:40.580 -> OK
16:45:40.580 ->
16:45:40.690 -> Ready for PING test
16:45:40.690 -> Command:
16:45:40.690 -> AT+PING="www.sequans.com"
16:45:45.694 -> No complete response received
16:45:45.694 ->
```

Figure 65: Response at red pin

6.3 Wi-Fi functionality

Figure 66 showcases the response in Serial Monitor after the Wi-Fi button was pressed, as well as the webpage with the ESP32's IP address entered as the URL. Wi-Fi connection and IP address were printed out upon powering up the system. See the last lines in Appendix A.11.



```
  if (digitalRead(wifi_output_point)) {
    return "checked";
  } else {
    return "";
  }
  return "";
}


//======================= VOID SETUP =======================//

void setup() {
  Serial.begin(BAUD_RATE);

  // Enable physical button as wake up source after ESP32 has entered light sleep mode
  pinMode(esp32_wake_BUTTON, INPUT_PULLUP);
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_33, 0);

  //------------------ NB-IoT setup ------------------//

  // Begin UART communication
  esp32Serial.begin(BAUD_RATE);    // Between ESP32 and PC
  monarchSerial.begin(BAUD_RATE);  // Between ESP32 and Monarch 2

  Serial.setRxBufferSize(0);  // Disable software serial buffer

  pinMode(NB_BUTTON, INPUT_PULLUP);  // Configure the button pin as input with internal pull-up resistor

  // Configure hardware flow control pins
  pinMode(RTS_pin, OUTPUT);
  pinMode(CTS_pin, INPUT);
  pinMode(RING_pin, INPUT);

  /*
   * Functions that can be executed on startup
   * Placed in order of necessary execution
   * Required: uartInitMonarch, configureOperationMode, connectNetwork
   * Rest is optional, choose between PSM and eDRX
  */
  uartInitMonarch();  // Configure UART between ESP32 and Monarch
  selectSIM();              // Select SIM slot
  // powerSIM();               // Power correct SIM slot
  // configureOperationMode();  // Configure Monarch Operation Mode
```

Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')

```
17:31:24.312 ->
17:31:24.312 -> Finished Power Saveing Mode initialization.
17:31:24.312 ->
17:31:25.934 -> Connecting to WiFi ...
17:31:26.932 -> Connecting to WiFi ...
17:31:27.931 -> Connecting to WiFi ...
17:31:28.935 -> Connecting to WiFi ...
17:31:29.904 -> Connecting to WiFi ...
17:31:29.904 -> 192.168.43.205
17:31:29.904 ->
17:32:32.555 -> 1
17:32:34.394 -> 0
17:32:38.480 -> 1
```

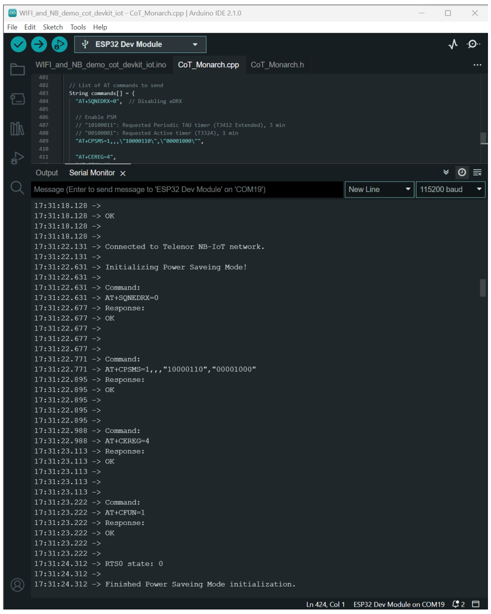
CoT DevKit IoT - WiFi demo

Figure 66: Wi-Fi button pressed with webpage

6.4 Monarch 2 power saving

Figure 67 present the last part of the AT commands needed to configure and enter Power Saving Mode (PSM), the function "initPSM()" was placed in the code's setup section; see Appendix A.11 for print out. Figure 61 shows that Monarch 2 woke up when the NB-IoT button was pressed and went to sleep after; see Appendix A.11 for printout.

Figure 68 depicts a picture of the LED connected to the PS_status/GPIO 2 pin during active mode. Even though cable was eliminated to reduce resistance, it did not illuminate.


WIFI_and_NB_demo_cot_devkit_iot - CoT_Monarch.cpp | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 424, Col 1
ESP32 Dev Module on COM19

```

  // List of AT commands to send
  String commands[] = {
    "AT+SQNEDRX=0",  // Disabling eDRX

    // Enable PSM
    // "10100011": Requested Periodic TAU timer (T3412 Extended), 3 min
    // "00100001": Requested Active timer (T3324), 1 min
    "AT+CPSMS=1,,,\"10000110\",\"00001000\"",

    "AT+CEREG=4",
```

```
17:31:18.128 ->
17:31:18.128 -> OK
17:31:18.128 ->
17:31:18.128 ->
17:31:22.131 -> Connected to Telenor NB-IoT network.
17:31:22.131 ->
17:31:22.631 -> Initializing Power Saveing Mode!
17:31:22.631 ->
17:31:22.631 -> Command:
17:31:22.631 -> AT+SQNEDRX=0
17:31:22.677 -> Response:
17:31:22.677 -> OK
17:31:22.677 ->
17:31:22.677 ->
17:31:22.677 ->
17:31:22.771 -> Command:
17:31:22.771 -> AT+CPSMS=1,,,"10000110","00001000"
17:31:22.895 -> Response:
17:31:22.895 -> OK
17:31:22.895 ->
17:31:22.895 ->
17:31:22.895 ->
17:31:22.988 -> Command:
17:31:22.988 -> AT+CEREG=4
17:31:23.113 -> Response:
17:31:23.113 -> OK
17:31:23.113 ->
17:31:23.113 ->
17:31:23.113 ->
17:31:23.222 -> Command:
17:31:23.222 -> AT+CFUN=1
17:31:23.222 -> Response:
17:31:23.222 -> OK
17:31:23.222 ->
17:31:23.222 ->
17:31:24.312 -> RTS0 state: 0
17:31:24.312 ->
17:31:24.312 -> Finished Power Saveing Mode initialization.
```

Figure 67: Power Saving Mode (PSM) initiated upon startup

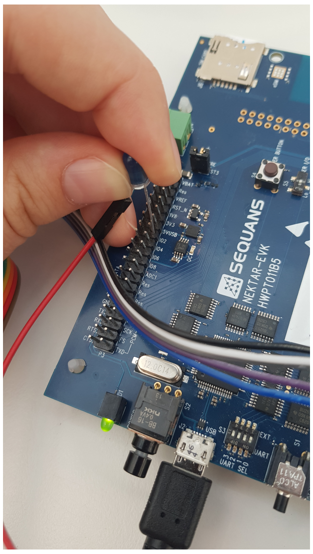
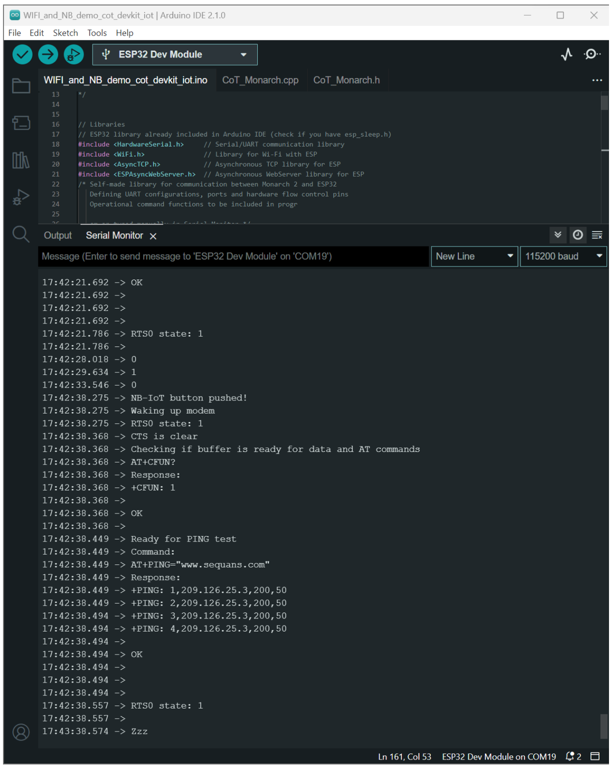


Figure 68: LED during active mode

6.5 ESP32 power saving

The Serial Monitor, as shown in Figure 69 below, displays the responses that occur after neither the Wi-Fi nor the NB-IoT buttons have been pressed within the last minute. Additionally, the Serial Monitor also displays the responses following the pressing of the ESP32 wake-up button. The "Zzz" indicates that the MCU went into light sleep mode. This is also printed out in the last lines of Appendix A.11.


WIFI_and_NB_demo_cot_devkit_iot | Arduino IDE 2.1.0
File Edit Sketch Tools Help
ESP32 Dev Module
WIFI_and_NB_demo_cot_devkit_iot.ino
CoT_Monarch.cpp
CoT_Monarch.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM19')
New Line
115200 baud
Ln 161, Col 53
ESP32 Dev Module on COM19
2

```
*/



// Libraries
// ESP32 library already included in Arduino IDE (check if you have esp_sleep.h)
#include <HardwareSerial.h>     // Serial/UART communication library
#include <WiFi.h>               // Library for Wi-Fi with ESP
#include <AsyncTCP.h>           // Asynchronous TCP library for ESP
#include <ESPAsyncWebServer.h>  // Asynchronous WebServer library for ESP
/* Self-made library for communication between Monarch 2 and ESP32
  Defining UART configurations, ports and hardware flow control pins
  Operational command functions to be included in progr

```

```
17:42:21.692 -> OK
17:42:21.692 ->
17:42:21.692 ->
17:42:21.692 ->
17:42:21.786 -> RTS0 state: 1
17:42:21.786 ->
17:42:28.018 -> 0
17:42:29.634 -> 1
17:42:33.546 -> 0
17:42:38.275 -> NB-IoT button pushed!
17:42:38.275 -> Waking up modem
17:42:38.275 -> RTS0 state: 1
17:42:38.368 -> CTS is clear
17:42:38.368 -> Checking if buffer is ready for data and AT commands
17:42:38.368 -> AT+CFUN?
17:42:38.368 -> Response:
17:42:38.368 -> +CFUN: 1
17:42:38.368 ->
17:42:38.368 -> OK
17:42:38.368 ->
17:42:38.449 -> Ready for PING test
17:42:38.449 -> Command:
17:42:38.449 -> AT+PING="www.sequans.com"
17:42:38.449 -> Response:
17:42:38.449 -> +PING: 1,209.126.25.3,200,50
17:42:38.449 -> +PING: 2,209.126.25.3,200,50
17:42:38.494 -> +PING: 3,209.126.25.3,200,50
17:42:38.494 -> +PING: 4,209.126.25.3,200,50
17:42:38.494 ->
17:42:38.494 -> OK
17:42:38.494 ->
17:42:38.494 ->
17:42:38.494 ->
17:42:38.557 -> RTS0 state: 1
17:42:38.557 ->
17:43:38.574 -> Zzz
```

Figure 69: ESP32 going to sleep and wake up button pressed

7 Analysis and discussion

This chapter provides an analysis of the results and the project's overall process and objective.

7.1 Project process

The main deviation from the original project plan was that the developer fell ill. This resulted in the original bachelor group, consisting of two people, to split. This turned out not to affect the project quality and proved to be a good solution. The original assignment was already quite divided, so this was not a problem. Consequently, this thesis was delayed by approximately two months, but all together, work hours and cost did not deviate much, as shown in the Gantt chart A.1 and S-curve A.2. This decision allowed the previous partner to complete their part on time and allowed this thesis's project developer to complete a sufficient product.

7.2 ESP32 and Monarch 2 compatibility

Manually typing in AT commands and function names in Serial Monitor and placing the functions within the code consistently worked. See Figures 58, 59, and 60, as well as Appendix A.14. The timing of the responses aligned with the delays programmed into the code. In some cases, it took the response longer than expected but eventually came later. Indicating that the system was able to catch up. However, if the response took too long, it would overwrite the next command and, as defined in the code, would abort further execution. Nevertheless, in most instances, the system functioned as intended.

The system did not work in some instances. It seemed to occur whenever the divide had been active for an extended period of time. A possible explanation for this could be data overflow on the UART line between Monarch 2 and ESP32, indicating that the buffer was full. Improving the hardware flow control could be a solution. The RING line on Monarch 2, which is responsible for monitoring data and unsolicited result codes (URCs) on the UART line, is handled internally. However, more comprehensive monitoring of the RING line and adapting ESP32's behavior accordingly could be a substantial improvement. Another suggestion for finer hardware flow control is to explore alternative UART libraries. Other hardware flow control libraries were tested but did not work. Therefore better monitoring of the UART line appears to be more efficient. The hardware flow control library utilised is HardwareSerial. This already includes the necessary technical specifications outlined in Monarch 2's documentation (B.5), and there is no need for a more complex library. It is important to note that if another more sufficient library is discovered in the future, it is possible to implement it since this is related to software and not hardware connections.

7.3 NB-IoT functionality

The NB-IoT button PING connectivity test generally worked well, see Figure 61 and Appendix A.12. However, for some instances when the ESP32 sent the request, Monarch 2 responded by not being ready to transfer data over the network. This could be because of wake-up latency after the ESP32 wake-up button was pushed since it worked consistently after a few seconds. The Monarch 2 latency is described in its documentation (B.5 page 27). Possible solutions could be to implement a longer system wake-up delay or implement a debounce on the button.

The SIM cards exhibited varying behaviour, where they sometimes worked and sometimes did not. The reasons for this happening align with the theory proposed earlier regarding buffer overflow in "7.2 ESP32 and Monarch 2 compatibility". Switching the SIM cards meant turning the entire system off and on, which could have cleared the buffer. Co-supervisor Nils Kristian Rossing, who had previous experience from a narrowband IoT project, encountered similar problems with Telenor SIM cards. His solution was to implement a watchdog timer (WDT). A watchdog could be used to monitor the CoT DevKit IoT. This approach is also suggested in Monarch 2's documentation (B.5 page 9). Rossing's watchdog operated in a more abrupt manner, completely shutting down and waking up the system. Something that is not suitable for this project, as the whole setup section of the code is then executed upon power-up. Based on these observations, it is likely that the issue lies within the system itself rather than the SIM cards and narrowband IoT network.

The initial plan was to test the NB-IoT connectivity with the IoT platform, Deploii, which is under development at Company of Things. However, at the time of testing for this project, the platform was not ready. Although it was possible to configure a combination of different HTTP AT command protocols from Monarch 2 to make the CoT DevKit IoT appear online on the platform, this approach was not prioritised. This is because of that it did not involve the transfer of sensor data, which was the original plan and a better demonstration of the system's functionality. Therefore the test switched to a PING. This test demonstrated that the system could connect to the network. However, it does not fully illustrate the system's intended functionality of the buttons simulating sensor data. Given the circumstances with Deploii, the PING test was deemed sufficient for demonstrating the CoT DevKit IoT's ability to connect to the narrowband network.

7.4 NB-IoT functionality inside and outside

There was virtually no difference between responses from testing inside Realfagsbygget at Gløshaugen NTNU and outside at Tømmerdalen, see Appendix A.12, as well as Figures 61 and 64. This demonstrates how well the narrowband IoT network works both inside and outside. This corresponds with the fact that narrowband IoT also should have good coverage inside compared to other cellular networking technologies such as 4G. The button was pressed along the line in the map in Figure 63. Connectivity was lost at the red pin and gained back at the purple pin. This matches well with the Telenor coverage map in Figure 62, making this a reliable map for students to refer to.

7.5 Wi-Fi

The Wi-Fi button demonstrated a high level of reliability, with minimal difficulties in updating the button state on the webpage when the button was pushed. See Figure 66. Only when the local Wi-Fi network encountered connectivity problems was the reliability of the connection compromised. The Wi-Fi button did include debounce, which the NB-IoT button did not. However, there was a delay when the ESP32 woke up, the same as with the NB-IoT button.

7.6 ESP32 power saving

As illustrated in Figure 69, the ESP32, and subsequently the entire system, entered light sleep mode after one minute and 17 milliseconds of inactivity from any of the buttons. This consistent delay of 10-20 milliseconds indicates a reliable transition into sleep mode. Something that could simply only be the delay of printing the message in Serial Monitor. Nonetheless, these measurements are within an acceptable time frame.

The system woke up when the ESP32 wake-up button was pressed, with approximately a 0,5-second delay, meaning that power-up was not instantaneous. Data/button presses before it was functional, was not registered. Nonetheless, this was not critical for this demonstration setup but could become an issue for other projects that require real-time data transfer. Temporarily storing the data might be a solution to prevent data loss.

7.7 Monarch 2 power saving

Feedback from Monarch 2 indicated that it went into power saving mode and sleep after setup and after the NB-IoT button was done sending, see Figure 61. This part went smoothly. The reason for placing the LED in the PS_status line was for students to visually see that the power states changed. However, as seen in Figure 68, this did not work. The voltage output on the pin should have been just enough to illuminate it, but it did not. With a multimeter, it was measured to be 1,7V when it should have been 1,8V. But it makes sense to have real-world differences. Read more about measuring Monarch 2's power consumption in the next subsection.

7.8 Measure Monarch 2 power consumption

If not for the scope of this project, it would have been beneficial to graph the different power states and consumption of Monarch 2, as well as measure the overall system's power consumption. However, this proved to be more challenging than initially anticipated.

Finding an oscilloscope with the required accuracy, as stated in Monarch 2's "Power Consumption Measurements on Monarch 2" document B.8, was difficult. It should be at par or under 1μA. The Department of Electronic Systems at NTNU had an oscilloscope that could meet the accuracy requirement, but there was not enough time to complete the second necessary step. Since Monarch 2 is a sensitive module, a protective circuit would be needed for power measurements. Senior engineer Ingulf Helland from the department recommended using the TC1262-5.0VDB Low-dropout regulator (LDO) from Microchip Technologies [87]. An LDO can regulate the output voltage even when the supply voltage is very close to the output voltage. The circuit depicted in Figure 70 would have been implemented if there was enough time to prioritise it. By knowing the input voltage and the consumption of the LDO with the variable resistor R3, it would have been possible to measure over R1 and calculate Monarch 2's power consumption. The C2 LDO output capacitor stabilises the LDO, and C1 stabilises the input. The R2 resistor acts as a dummy-proof component in case the variable resistor is turned to 0Ω.

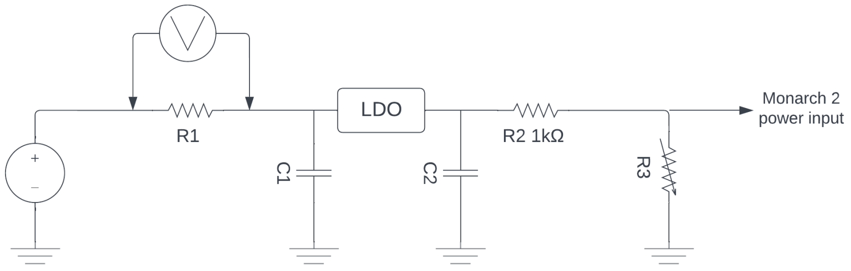


Figure 70: Monarch 2 protective power measurement circuit

7.9 Product cost and market value

As of July 2023, the Monarch 2 GM02S module is priced at around 200 NOK, depending on the quantity purchased [88]. On the other hand, the ESP32-WROOM-32E module is priced at 32 NOK [89]. Considering the system requirement of achieving low cost, the fact that these two main components only amount to a total of 232 NOK is very favourable.

When considering the market value of the development board with both Wi-Fi and NB-IoT capabilities in March 2023, the LILYGO T-SIM7080G-S3 emerged as a competitor with a price of 320 NOK, as seen in Table 3 in Chapter 3.4.1. However, it lacked sufficient support documentation and reliable distributors. Another option was the combination of Wappsto:bit and Micro:bit, which amounted to approximately 970 NOK (Figure 4). This was both expensive and impractical, as it required two separate boards. The Raspberry Pi Pico W combined with the SIM7020E NB-IoT module was also considered as a solution to integrate all desired functionalities but came with the same impracticality of being two separate boards.

However, during the course of this project, a competitor emerged. QuickSpot is developing a board called "Walter" with ESP32-S3 and Monarch 2 [90]. This presents significant competition, as it utilised the same components, albeit with ESP32-S3 instead of ESP32. QuickSpot appears to have a larger team dedicated to its development. From a positive perspective, this could indicate that the selection of the main components was a good choice. However, Company of Things may need to reassess CoT DevKit IoT. One possibility is to explore the option of switching to the TX82 NB-IoT modem, which was a close second to the Monarch 2. Although it is slightly more expensive, it could be a viable alternative. Additionally, Company of Things could focus on specifically targeting the Norwegian market and offer package deals to schools, leveraging their IoT platform Deploii. This aligns with the goal of integrating CoT DevKit IoT with the Deploii platform and offering software solutions as an added advantage. The bachelor thesis project developer recommends Company of Things use this strategic marketing approach rather than substitute the components.

7.10 Outline example of student project assignment

An outline of a student assignment can be created by considering the project results and the pedagogical principles related to the development board, which are elaborated on in Chapter "2.6 Pedagogical principles". This example is not only limited to the finalised CoT DevKit IoT board, as the proof-of-concept has demonstrated usability at this level. However, continuous improvement in its functionality can make it even better. The development board enables teachers to design project-based assignments that provide students with active and engaging hands-on learning experiences. Dividing into groups allows them to acquire communication skills. For enhanced comprehension, the assignment can be contextual and problem-centered, with teachers presenting examples of how IoT technology can solve real-world problems. The project may have a somewhat predefined solution if the objective is for students to learn something specific, or it can remain open-ended to allow students to genuinely solve the task on their own. The CoT DevKit IoT is versatile and suitable for a diverse range of projects. Below is an example that takes everything mentioned into account.

Title/Problem to solve: Where is it best to place solar panels on campus?
Group: 2-3 students
Components:
CoT DevKit IoT, PC, LDR/photoresistor (light sensors), battery, wires and resistors
Description:
The school wants to install solar panels on campus, and it could be exciting to involve students in the process. Each group places the set up at different locations on campus,

lasting from, for example, a day to a week. They will collect light data using either Wi-Fi and/or NB-IoT, depending on which they decide suits the purpose better. If possible, data collection can be done at different times during the school year to take seasons changing into consideration. This approach might not be the most accurate, but this can also provide a valuable starting point for discussion among students. The mentioned components represent the basic requirements, but students are encouraged to explore further possibilities. The components mentioned are only the basics, meaning that students can, for example, add more, create a capsule, or whatever they deem fit to address the problem effectively.

This project fosters problem-solving skills, critical thinking, creativity, and collaboration.

7.11 Future work

The design produce of a development board is a complex and time-consuming process. Based on the project findings, the following list is areas that Company of Things should prioritise for future work:

- Improved UART flow control: Enhance hardware flow control between Monarch 2 and ESP32. This is to prevent data overflow and improve overall reliability by ESP32 monitoring the RING line more and implementing debounce on the button.
- Watchdog implementation: Integrating a watchdog timer into the system to monitor and control the operation of the ESP32 and Monarch 2, ensuring stability and preventing potential issues.
- Power consumption measurement: Conducting power consumption measurements on the Monarch 2 module using an accurate oscilloscope and implementing the protective circuit for safe measurements. To do this, Company of Things would need to invest in an accurate enough oscilloscope.
- External power source: Implement external power source, such as batteries with a battery management system and the mentioned watchdog.
- Alternative NB-IoT modem evaluation: Assessing the possibility of using alternative NB-IoT modems, such as the TX82, which may offer improved features or performance compared to the Monarch 2.
- Integrate with Deploii: Establish connectivity and data transfer over the NB-IoT network between CoT DevKit IoT and the IoT platform Deploii when Deploii is ready.
- Expansion of functionalities and reliability testing: The addition of new functionalities or sensors to the CoT DevKit IoT, to expand its capabilities and versatility for various IoT applications. This translates to conducting extensive reliability testing, including stress testing, to identify any potential weaknesses or issues in the system and address them accordingly. This becomes even more important when operation with Deploii is established.

- Test with students: Conducting testing of the proof-of-concept with students would provide valuable insights into potential challenges and perspectives that may not have been considered. Additionally, fostering further collaboration and cooperation with teachers and lecturers would be a wise step to gather feedback and enhance the project's effectiveness.
- Expand to Bluetooth and 4G: The main components already have these functionalities, and expanding the system characteristics makes it an even more well-rounded IoT development board.

8 Conclusion

The problem statement/project goal provided by Company of Things was to: "Create a proof-of-concept of an adaptable Internet of Things (IoT) development board, customised to cater to the specific demands and needs of the Norwegian education system". This translated to the functionality requirements they also desired. The board needed to be suitable for both short-range indoor and long-range outdoor usage to facilitate learning about IoT in different environments. The programming language needed to be beginner-friendly, as the board is going to be used in entry-level courses. Low-power consumption is a central aspect of IoT. This was, therefore, an attribute they wanted. Lastly, keeping the main components at a low cost was important due to limited financial resources within the educational system. Throughout this project process, the development of the CoT DevKit IoT proof-of-concept has been carried out with consideration to these requisites.

These functionality requirements were translated into deciding on specific technological system requirements to solve the problem statement. The choices were based on acquired information from teachers at the Technology and Research Education Conference and extensive online research. Wi-Fi was selected for short-range indoor use, while narrowband IoT was chosen for long-range outdoor use. Arduino C was selected as the programming language as it is suitable for beginners in microcontroller programming, and Serial Monitor in its IDE is useful for users. To meet these requirements, achieve possible low-power consumption, and strive for low cost, the main components chosen were the Monarch 2 GMS02S module that works as the NB-IoT modem and the ESP32-WROOM-32E microcontroller module with integrated Wi-Fi that operates as the system's main processing unit. All these system requirements were met but could have been expanded upon to improve these functionalities even more. This bachelor project is the beginning of the development of CoT DevKit IoT, which is a long process. Therefore are, the project results excellent, and the final proof-of-concept is well-suited for education.

The finalised system in this project demonstrated the different functionalities of CoT DevKiT IoT. It consisted of three buttons to prove Wi-Fi, NB-IoT, and low-power consumption functionalities. When the "Wi-Fi button" was pressed, its state was successfully updated on a webpage hosted by the ESP32, proving the system's Wi-Fi capabilities. When the "NB-IoT button" was pressed, it woke Monarch 2 up from sleep and sent a PING test to a website to confirm a network connection. The narrowband test worked both inside and outside, matching the Telenor coverage map. To save even more energy, after a minute of inactivity, the ESP32 went into light sleep. The ESP32 was successfully woken up by pressing the "ESP32 wake up button". The Monarch 2 and ESP32 proved to be excellently compatible and easy to use with Arduino C. The ESP32 could control Monarch 2 with AT commands being automatically executed and implemented in the code, or users could manually type them in. Some areas for improvement were identified for communication between the two components and data transfer on the NB-IoT network. The buffers seemed to overflow. To fix this, improved dataflow monitoring and control should be implemented. Furthermore, the original goal was to connect the board to Company of Things' IoT platform, Deploii. This could not be achieved due to the platform not being ready for NB-IoT data transactions. The PING test served as a sufficient replacement, even though data transfer could not be properly tested.

Some unforeseen challenges occurred throughout the project. The project developer fell ill, which led to a delay of two months. Despite this, a satisfactory product was presented at the end. The original estimated cost of this project, including time and resources, was 177 650 NOK, and the actual cost ended up to be 181 497 NOK. As of March 2023, there was no substantial competitor. However, Walter from QuickSpot was later discovered. This board is not jet available on the market but is set to employ similar components. For this reason, other components might be worth considering. Nevertheless, Monarch 2 and ESP32 seem to be the most suitable components, and the recommendation is, therefore, not to switch them. The recommended strategy from the project developer is to tailor CoT DevKit IoT even more to the Norwegian market. This encompasses the fact that convenience and the easiest implementation are what schools desire and have the capacity for. Offering a package deal with both Deploii and CoT DevKit IoT could be an effective plan of action.

Developing a development board is a long process. After this bachelor project conclusion, there are some things to prioritise for future work. Detailed elaboration can be found in Chapter "7.11 Future work". The first thing Company of Things should do is to assess if they actually want to use these components further, even though they are recommended by this bachelor project developer. The second thing is to improve monitoring of the Ring line to hopefully improve UART flow control. If this does not help, the next step is to employ a watchdog. Furthermore, implement external power sources and measure power consumption. Once Deploii is ready, data transfer with the platform should be prioritised. The demonstration could be expanded upon with different sensors and purposes to test the board's versatility. Having trials with students would be valuable. Exploring the already built-in Bluetooth and 4G functionalities in the main components could be led to an even more versatile IoT board. Only after all these improvements should Company of Things proceed with designing the printed circuit board (PCB) layout.

In conclusion, the CoT DevKit IoT proof-of-concept with Monarch 2 and ESP32 met all functionality requirements to be well suited for usage in the Norwegian educational system. Wi-Fi and NB-IoT allow for versatile IoT projects in different environments. Arduino C and Arduino IDE with Serial Monitor facilitate beginner-friendliness. Its focus on low power demonstrates this central IoT concept for students. The cost-effectiveness of the main components is an important achievement. There are areas of improvement, and its market value must be acknowledged. However, the proof-of-concept exhibits a robust groundwork for further development of CoT DevKit IoT.

9 References

[1] A. S. Gillis, *What is the internet of things (iot)?* Available at https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT (March 2022).

[2] IBM, *What is industry 4.0?* Available at https://www.ibm.com/topics/industry-4-0 (2023).

[3] T. Engineering, *Internet of things (iot) in engineering*, Available at https://technosofteng.com/applications-of-internet-of-things-iot-in-engineering/ (28.07.2020).

[4] A. Garg, *How to connect iot sensors wirelessly with a web application?* Available at https://www.analyticsvidhya.com/blog/2022/09/how-to-connect-iot-sensors-wirelessly-with-a-web-application/ (02.09.2022).

[5] B. Lutkevich, *Microcontroller (mcu)*, Available at https://www.techtarget.com/iotagenda/definition/microcontroller (November 2019).

[6] BBC, *Systems approach to designing*, Available at https://www.bbc.co.uk/bitesize/guides/z6kr97h/revision/3 (2023).

[7] M. Brain and T. Homer, *How wifi works*, Available at https://computer.howstuffworks.com/wireless-network.htm (17.08.2021).

[8] L. Rosencrance, *Narrowband iot (nb-iot)*, Available at https://www.techtarget.com/whatis/definition/narrowband-IoT-NB-IoT (2023).

[9] Telenor, *Lte-m vs nb-iot – a guide exploring the differences between lte-m and nb-iot*, Available at https://iot.telenor.com/iot-insights/lte-m-vs-nb-iot-guide-differences/ (2023).

[10] A. Froehlich, *What's the role of narrowband iot in 5g networks?* Available at https://www.techtarget.com/searchnetworking/answer/Whats-the-role-of-narrowband-IoT-in-5G-networks (2023).

[11] N. Agnihotri, *At commands, gsm at command set*, Available at https://www.engineersgarage.com/at-commands-gsm-at-command-set/ (2023).

[12] Particle, *An introduction to low power iot*, Available at https://www.particle.io/iot-guides-and-resources/low-power-iot/ (2023).

[13] J. Tollefson and L. Reese, *Low power technology*, Available at https://no.mouser.com/applications/low-power-ewc-low-power/ (2023).

[14] 1NCE, *Psm and edrx: Power saving in cellular lpwan - possibilities and limitations*, Available at https://1nce.com/en-eu/resources/news-insights/blog/psm-and-edrx (2023).

[15] Velos, *What are psm and edrx features in lte-m and nb-iot?* Available at https://blog.velosiot.com/what-are-psm-edrx-features-in-lte-m-and-nb-iot (16.03.2022).

[16] N. S. of Architecture and Design, *What are the benefits of hands-on learning?* Available at https://newschoolarch.edu/blog/what-are-the-benefits-of-hands-on-learning/ (2023).

[17] C. Pritchett, *What is contextual learning*, Available at https://www.igi-global.com/dictionary/contextual-learning/5675 (2008).

[18] U. of Illinois at Urbana-Champaign Center for Innovation in Teaching Learning, *Problem-based learning (pbl)*, Available at https://citl.illinois.edu/citl-101/teaching-learning/resources/teaching-strategies/problem-based-learning-(pbl) (2023).

[19] I. Andreev, *Collaborative learning*, Available at https://www.valamis.com/hub/collaborative-learning (21.06.2023).

[20] A. Norris, *Flexibility and the project approach*, Available at https://illinoisearlylearnin org/blogs/perspectives/flexibility-project/ (2023).

[21] E. T. Garvik, *Utdanning.no: Elektroingeniør*, Available at https://utdanning.no/tema/yrkesintervju/elektroingenior (2023).

[22] B. Technology, *Bluetooth specifications*, Available at https://www.bluetooth.com/specifications/ (2023).

[23] S. M. Kerner, *4g (fourth-generation wireless)*, Available at https://www.techtarget.com/searchmobilecomputing/definition/4G (01.04.2021).

[24] M. Company, *What is 5g?* Available at https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-5g (07.10.2021).

[25] Semtech, *What is lora?* Available at https://www.semtech.com/lora/what-is-lora (2023).

[26] E. Helium, *Lorawan hotspot map*, Available at https://explorer.helium.com/ (2023).

[27] Telenor, *Dekningskart*, Available at https://www.telenor.no/dekning/#dekningskart (2023).

[28] L. Xiao, *What is c++ used for?* Available at https://www.codecademy.com/resources/blog/what-is-c-plus-plus-used-for/ (10.05.2021).

[29] S. Shinde, *What are the key pros and cons of the arduino programming language?* Available at https://emeritus.org/blog/coding-arduino-programming-language/ (25.01.2023).

[30] A. Subero, *Programming Microcontrollers with Python*, 1st ed. Apress, 2021.

[31] Arduino, *Arduino ide downloads*, Available at https://www.arduino.cc/en/software (2023).

[32] G2, *Best terminal emulator software*, Available at https://www.g2.com/categories/terminal-emulator (2023).

[33] D. Yatsenko, *Try, then try again: Why iterative design process brings the finest results*, Available at https://www.eleken.co/blog-posts/iterative-design-process (2023).

[34] Lilygo, *T-sim7080g-s3*, Available at https://www.lilygo.cc/products/t-sim7080-s3 (2023).

[35] E. Systems, *Esp32-s3*, Available at https://www.espressif.com/en/products/socs/esp32-s3 (2023).

[36] Arduino, *Arduino mkr nb 1500*, Available at https://store.arduino.cc/products/arduino-mkr-nb-1500 (2023).

[37] M. Technology, *Avr-iot cellular mini*, Available at https://www.microchip.com/en-us/development-tool/ev70n78a (2023).

[38] SparkFun, *Sparkfun thing plus - esp32 wroom (micro-b)*, Available at https://www.sparkfun.com/products/15663 (2023).

[39] Arduino, *Arduino uno wifi rev2*, Available at https://store.arduino.cc/products/arduino-uno-wifi-rev2 (2023).

[40] DigiKey, *Esp32-devkitc-32e*, Available at https://www.digikey.no/no/products/detail/espressif-systems/ESP32-DEVKITC-32E/12091810 (2023).

[41] R. Pi, *Buy a raspberry pi pico*, Available at https://www.raspberrypi.com/products/raspberry-pi-pico/ (2023).

[42] SparkFun, *Sparkfun lte cat m1/nb-iot shield - sara-r4*, Available at https://www.sparkfun.com/products/14997 (2023).

[43] S. Studio, *Dragino nb-iot shield-b5*, Available at https://www.seeedstudio.com/Dragino-NB-IoT-Shield-B5.html (2023).

[44] M. Elektronika, *Nb iot click*, Available at https://www.mikroe.com/nb-iot-click (2023).

[45] Waveshare, *Sim7020e nb-iot hat*, Available at https://www.waveshare.com/wiki/SIM7020E_NB-IoT_HAT (2023).

[46] RS, *Wappsto:bit nb iot*, Available at https://no.rs-online.com/web/p/bbc-micro-bit-add-ons/2251594 (2023).

[47] SparkFun, *Sparkfun wifi shield - esp8266*, Available at https://www.sparkfun.com/products/13287 (2023).

[48] M. Elektronika, *Wifi 7 click*, Available at https://www.mikroe.com/wifi-7-click (2023).

[49] ——, *Wifi esp click*, Available at https://www.mikroe.com/wifi-esp-click (2023).

[50] SparkFun, *Micro:bit v2 board*, Available at https://www.sparkfun.com/products/17287 (2023).

[51] M. Electronics, *Esp32-s2-solo-n4r2*, Available at https://shorturl.at/xLPS6 (2023).

[52] ——, *Esp32-wroom-32e-n8*, Available at https://shorturl.at/zLNPV (2023).

[53] CC3220SF-LAUNCHXL, *How wifi works*, Available at https://shorturl.at/luWY1 (2023).

[54] M. Electronics, *Wfi32e01ue-i*, Available at https://shorturl.at/hiVZ3 (2023).

[55] R. Sheldon, *Sram (static random access memory)*, Available at https://www.techtarget.com/whatis/definition/SRAM-static-random-access-memory (2023).

[56] M. Electronics, *Esp32-devkitc v4 development board*, Available at https://no.mouser.com/new/espressif/espressif-esp32-devkitc-da-development-board/ (2023).

[57] E. Systems, *Esp32-wroom-32 datasheet*, Available at https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf (2023).

[58] ——, *Esp32-devkitc v4 getting started guide*, Available at https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html (2023).

[59] T. Electronics, *Bc66*, Available at https://www.top-electronics.com/en/bc66-multi-band-nb-iot-module-1 (2023).

[60] ——, *Bc95-gv*, Available at https://www.top-electronics.com/en/compact-nb-iot-module-multiband-1 (2023).

[61] T. Cinterion, *Exs82*, Available at https://www.telit.com/devices/exs82/ (2023).

[62] M. Electronics, *Lbad0zz1se-743*, Available at https://shorturl.at/esX56 (2023).

[63] Sequans, *Monarch 2 gm02s module (global)*, Available at https://sequans.com/products/monarch-2-gm02s/ (2023).

[64] M. Electronics, *Sara-r412m-02b*, Available at https://shorturl.at/foP04 (2023).

[65] SIMcom, *Sim7022*, Available at https://www.simcom.com/product/SIM7022.html (2023).

[66] T. Cinterion, *Tx82*, Available at https://www.telit.com/devices/tx82/ (2023).

[67] Sequans, *Monarch 2 gm02s nektar evaluation kit (global)*, Available at https://sequans.com/products/monarch-2-gm02s-nektar-evk/ (2023).

[68] ITpedia, *Hva er integrasjonstesting og hvorfor gjør vi det?* Available at https://no.itpedia.nl/2019/05/17/wat-is-integratietesten-en-waarom-doen-we-het/ (2023).

[69] Yuhiro, *Hva er enhetstesting*, Available at https://www.software-developer-india.com/no/hva-er-enhetstesting/ (2023).

[70] GeeksForGeeks, *Difference between dte and dce*, Available at https://www.geeksforgeeks.org/difference-between-dte-and-dce/ (2023).

[71] M. Technologies, *Hardware flow control*, Available at https://onlinedocs.microchip.com/pr/GUID-167CA20A-2C0F-4CBC-A693-9FD032B9B193-en-US-1/index.html?GUID-C8B83E54-0F62-4205-98DD-B1560AACDBB4 (2023).

[72] Riot, *At (hayes) command set library*, Available at https://doc.riot-os.org/group__drivers__at.html (2023).

[73] S. Santos, *How to set an esp32 access point (ap) for web server*, Available at https://randomnerdtutorials.com/esp32-access-point-ap-web-server/ (2018).

[74] M. Brain and T. Homer, *How wifi works*, Available at https://source.android.com/docs/core/connect/wifi-softap (2023).

[75] T. J., *What is a web server? how it works and more*, Available at https://www.hostinger.com/tutorials/what-is-a-web-server (07.06.2023).

[76] Cloudflare, *What is http?* Available at https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/ (2023).

[77] E. Systems, *Sleep modes*, Available at https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/sleep_modes.html (2023).

[78] L. M. Engineering, *Insight into esp32 sleep modes their power consumption*, Available at https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/ (2023).

[79] Telenor, *Strømbesparende funksjoner for lte-m og nb-iot*, Available at https://www.telenor.no/bedrift/iot/teknologi/edrx-spesifikasjoner/ (2023).

[80] R. N. Tutorials, *Installing the esp32 board in arduino ide (windows, mac os x, linux)*, Available at https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/ (2023).

[81] R. Santos, *Esp32 web server – arduino ide*, Available at https://randomnerdtutorials.com/esp32-web-server-arduino-ide/ (2023).

[82] E. Systems, *Esp-idf: Get started*, Available at https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/ (2023).

[83] ———, *Esp-idf: Universal asynchronous receiver/transmitter (uart)*, Available at https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/uart.html (2023).

[84] ———, *Esp-at: Hardware connection*, Available at https://docs.espressif.com/projects/esp-at/en/release-v2.2.0.0_esp32c3/Get_Started/Hardware_connection.html (2023).

[85] E. S. GitHub, *Hardwareserial.h*, Available at https://github.com/espressif/arduino-esp32/blob/master/cores/esp32/HardwareSerial.h (2018).

[86] T. Term, *Tera term home page*, Available at https://ttssh2.osdn.jp/index.html.en (2023).

[87] M. Electronics, *Tc1262-5.0vdb*, Available at https://shorturl.at/vwFUY (2023).

[88] Digikey, *Gm02s*, Available at https://shorturl.at/nARUX (2023).

[89] ——, *Esp32-wroom-32e-n4*, Available at https://shorturl.at/xzM45 (2023).

[90] Quickspot, *Meet walter, your new best friend*, Available at https://www.quickspot.io/ (2023).

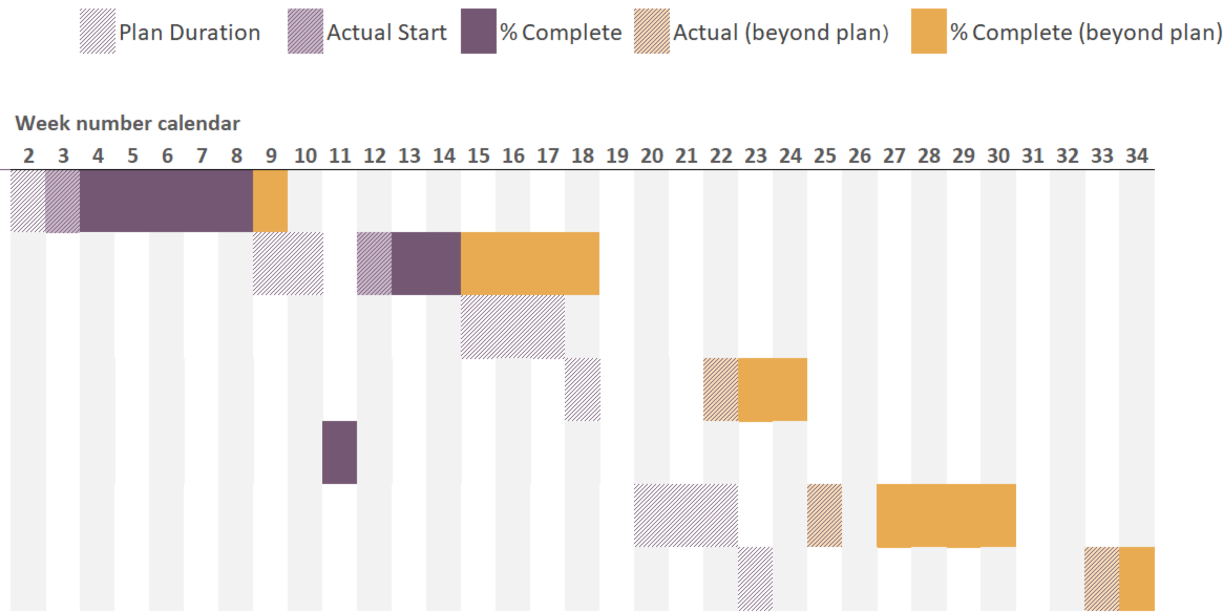
A Appendices

A.1 Gantt-chart

Gantt-chart of the bachelor project.

Final Gantt-chart bachelor

Work package	Plan start	Plan duration	Actual start	Actual duration	Complete [%]
H1	2	7	3	7	100%
H2	8	5	9	7	100%
H3	9	5	Never	0	0%
H4	18	1	22	3	100%
OF1	11	1	11	1	100%
P1	17	3	25	5	100%
P2	23	1	30	2	100%



A.2 S-curve

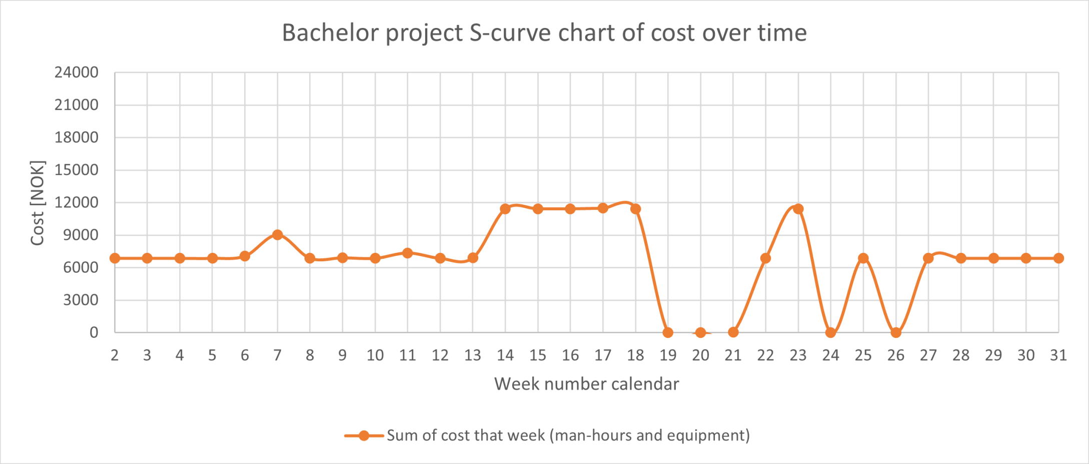


Figure 71: S-chart of the bachelor project

A.3 Survey Technology and Research Education Conference

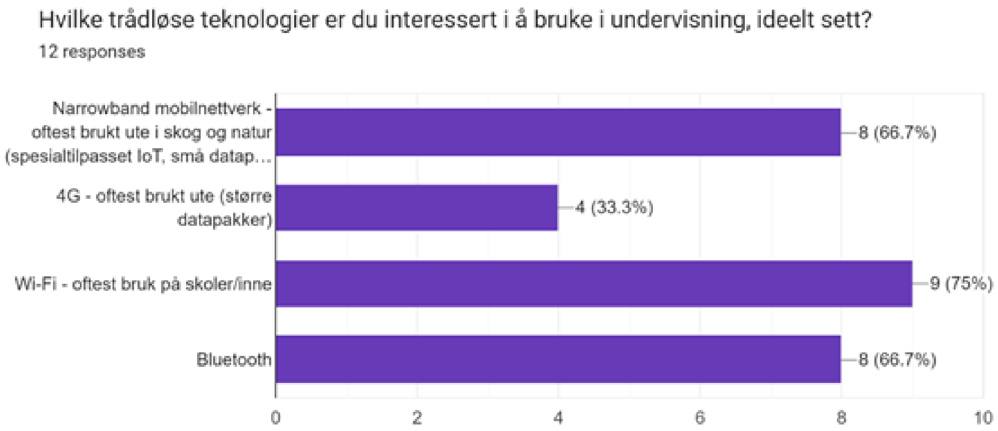


Figure 72: Survey question 1: "Which wireless technologies are you interested in utilising for educational purposes, ideally?

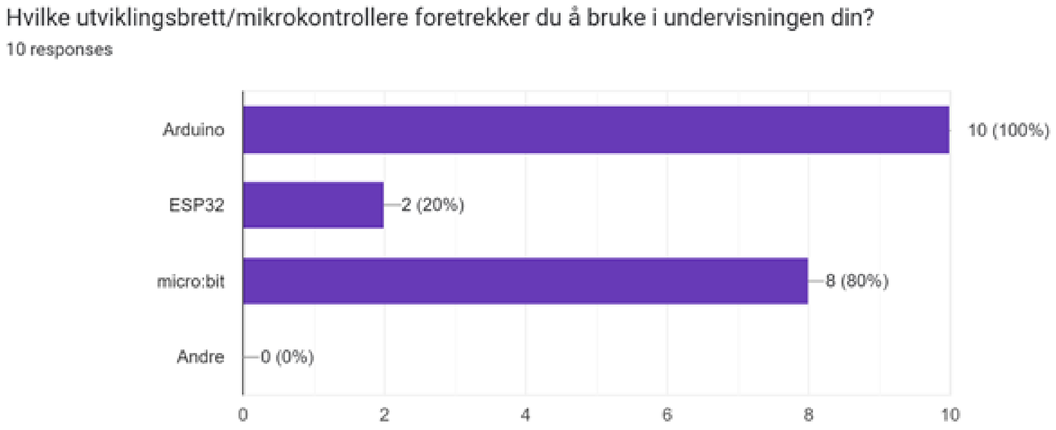


Figure 73: Survey question 2: "Which development boards/microcontrollers do you prefer to use in your teaching?"

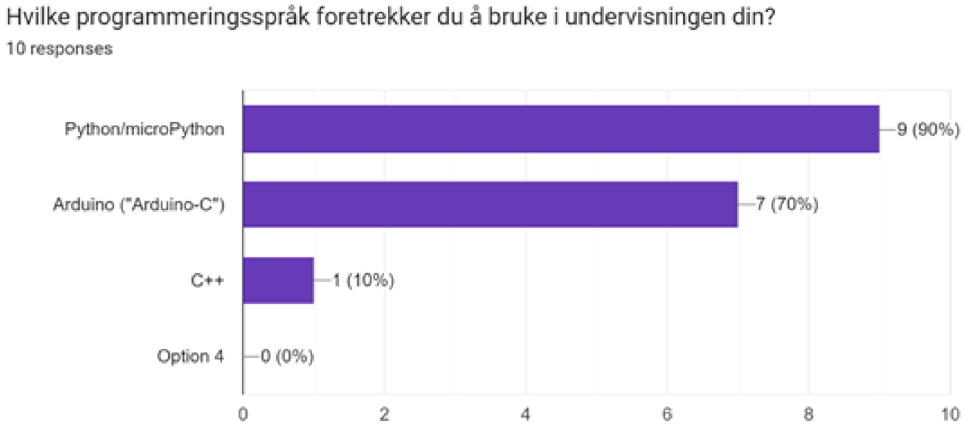


Figure 74: Survey question 3: "Which programming languages do you prefer to use in your teaching?"

A.4 ESP32 DevKit V4 pinout diagram

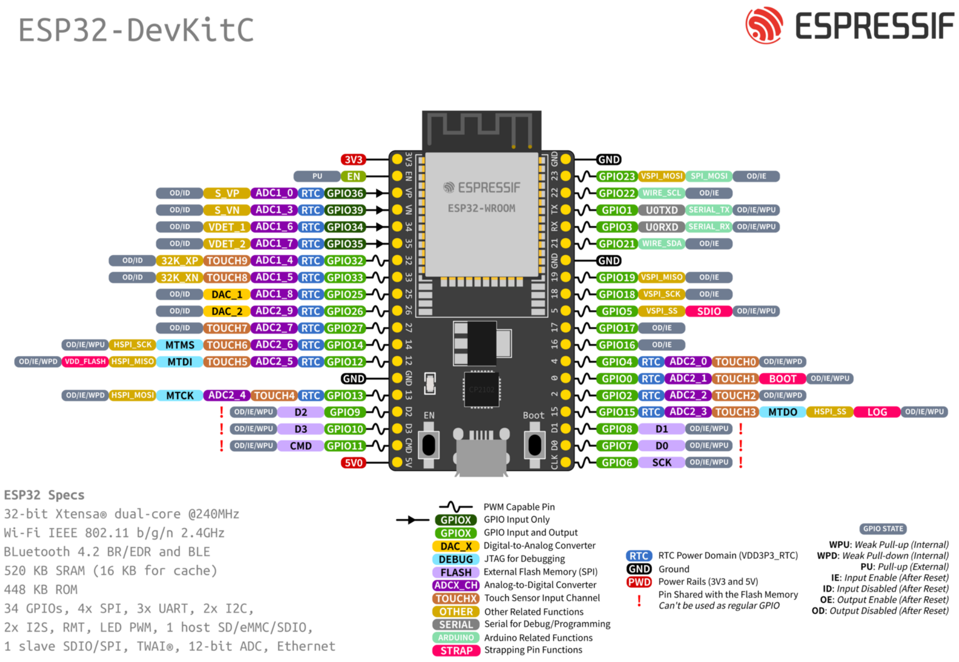

ESP32-DevKitC
ESPRESSIF
ESP32-WROOM
3V3
PU EN
OD/ID S_VP ADC1_0 RTC GPIO36
OD/ID S_VN ADC1_3 RTC GPIO39
OD/ID VDET_1 ADC1_6 RTC GPIO34
OD/ID VDET_2 ADC1_7 RTC GPIO35
OD/ID 32K_XP TOUCH9 ADC1_4 RTC GPIO32
OD/ID 32K_XN TOUCH8 ADC1_5 RTC GPIO33
OD/ID DAC_1 ADC1_8 RTC GPIO25
OD/ID DAC_2 ADC2_9 RTC GPIO26
OD/ID TOUCH7 ADC2_7 RTC GPIO27
OD/IE/WPU HSPI_SCK MTMS TOUCH6 ADC2_6 RTC GPIO14
OD/IE/WPD VDD_FLASH HSPI_MISO MTDI TOUCH5 ADC2_5 RTC GPIO12
GND
OD/IE/WPD HSPI_MOSI MTCK ADC2_4 TOUCH4 RTC GPIO13
! OD/IE/WPU D2 GPIO9
! OD/IE/WPU D3 GPIO10
! OD/IE/WPU CMD GPIO11
5V0
3V3 EN VP VN 34 35 32 33 25 26 27 14 12 GND 13 D2 D3 CMD 5V
GND 23 22 TX RX 21 GND 19 18 5 17 16 4 0 2 15 D1 D0 CLK
EN
Boot
GND
GPIO23 VSPI_MOSI SPI_MOSI OD/IE
GPIO22 WIRE_SCL OD/IE
GPIO1 U0TXD SERIAL_TX OD/IE/WPU
GPIO3 U0RXD SERIAL_RX OD/IE/WPU
GPIO21 WIRE_SDA OD/IE
GND
GPIO19 VSPI_MISO OD/IE
GPIO18 VSPI_SCK OD/IE
GPIO5 VSPI_SS SDIO OD/IE/WPU
GPIO17 OD/IE
GPIO16 OD/IE
GPIO4 RTC ADC2_0 TOUCH0 OD/IE/WPD
GPIO0 RTC ADC2_1 TOUCH1 BOOT OD/IE/WPU
GPIO2 RTC ADC2_2 TOUCH2 OD/IE/WPD
GPIO15 RTC ADC2_3 TOUCH3 MTDO HSPI_SS LOG OD/IE/WPU
GPIO8 D1 OD/IE/WPU !
GPIO7 D0 OD/IE/WPU !
GPIO6 SCK OD/IE/WPU !
ESP32 Specs
32-bit Xtensa® dual-core @240MHz
Wi-Fi IEEE 802.11 b/g/n 2.4GHz
BLuetooth 4.2 BR/EDR and BLE
520 KB SRAM (16 KB for cache)
448 KB ROM
34 GPIOs, 4x SPI, 3x UART, 2x I2C,
2x I2S, RMT, LED PWM, 1 host SD/eMMC/SDIO,
1 slave SDIO/SPI, TWAI®, 12-bit ADC, Ethernet
PWM Capable Pin
GPIOX GPIO Input Only
GPIOX GPIO Input and Output
DAC_X Digital-to-Analog Converter
DEBUG JTAG for Debugging
FLASH External Flash Memory (SPI)
ADCX_CH Analog-to-Digital Converter
TOUCHX Touch Sensor Input Channel
OTHER Other Related Functions
SERIAL Serial for Debug/Programming
ARDUINO Arduino Related Functions
STRAP Strapping Pin Functions
RTC RTC Power Domain (VDD3P3_RTC)
GND Ground
PWD Power Rails (3V3 and 5V)
! Pin Shared with the Flash Memory Can't be used as regular GPIO
GPIO STATE
WPU: Weak Pull-up (Internal)
WPD: Weak Pull-down (Internal)
PU: Pull-up (External)
IE: Input Enable (After Reset)
ID: Input Disabled (After Reset)
OE: Output Enable (After Reset)
OD: Output Disabled (After Reset)

Figure 75: ESP32 pinout diagram [58]

A.5 Monarch 2 pinout sockets

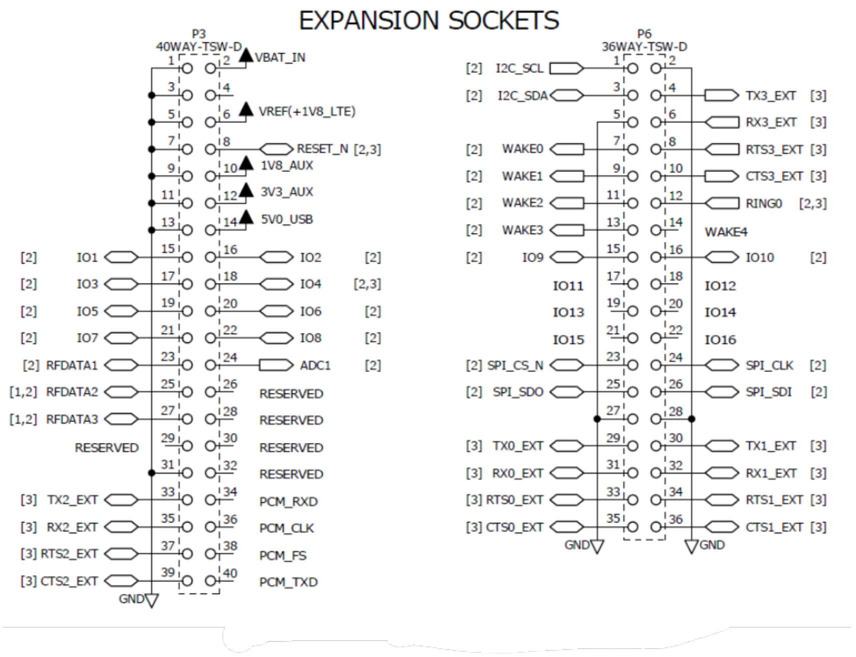

EXPANSION SOCKETS
P3
40WAY-TSW-D
1 2 VBAT_IN
3 4
5 6 VREF(+1V8_LTE)
7 8 RESET_N [2,3]
9 10 1V8_AUX
11 12 3V3_AUX
13 14 5V0_USB
[2] IO1 15 16 IO2 [2]
[2] IO3 17 18 IO4 [2,3]
[2] IO5 19 20 IO6 [2]
[2] IO7 21 22 IO8 [2]
[2] RFDATA1 23 24 ADC1 [2]
[1,2] RFDATA2 25 26 RESERVED
[1,2] RFDATA3 27 28 RESERVED
RESERVED 29 30 RESERVED
31 32 RESERVED
[3] TX2_EXT 33 34 PCM_RXD
[3] RX2_EXT 35 36 PCM_CLK
[3] RTS2_EXT 37 38 PCM_FS
[3] CTS2_EXT 39 40 PCM_TXD
GND
P6
36WAY-TSW-D
[2] I2C_SCL 1 2
[2] I2C_SDA 3 4 TX3_EXT [3]
5 6 RX3_EXT [3]
[2] WAKE0 7 8 RTS3_EXT [3]
[2] WAKE1 9 10 CTS3_EXT [3]
[2] WAKE2 11 12 RING0 [2,3]
[2] WAKE3 13 14 WAKE4
[2] IO9 15 16 IO10 [2]
IO11 17 18 IO12
IO13 19 20 IO14
IO15 21 22 IO16
[2] SPI_CS_N 23 24 SPI_CLK [2]
[2] SPI_SDO 25 26 SPI_SDI [2]
27 28
[3] TX0_EXT 29 30 TX1_EXT [3]
[3] RX0_EXT 31 32 RX1_EXT [3]
[3] RTS0_EXT 33 34 RTS1_EXT [3]
[3] CTS0_EXT 35 36 CTS1_EXT [3]
GND GND

Figure 76: Pinout of expansion connections on Monarch 2 EVK (B.3 page 22)

A.6 ESP32 block diagram

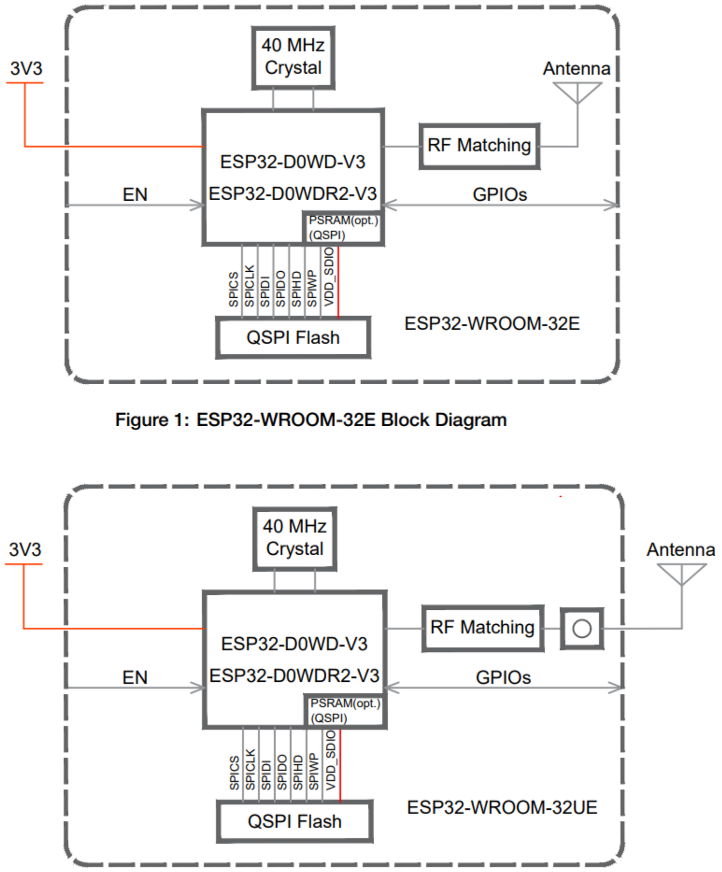


Figure 77: ESP32-WROOM-32 block diagram [57]

A.7 Monarch 2 block diagram

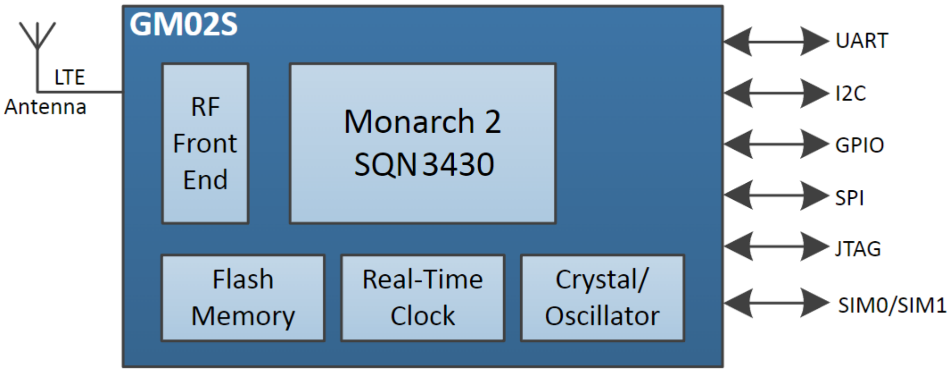


Figure 78: Monarch 2 block diagram (B.2 page 2)

A.8 ESP32 schematics

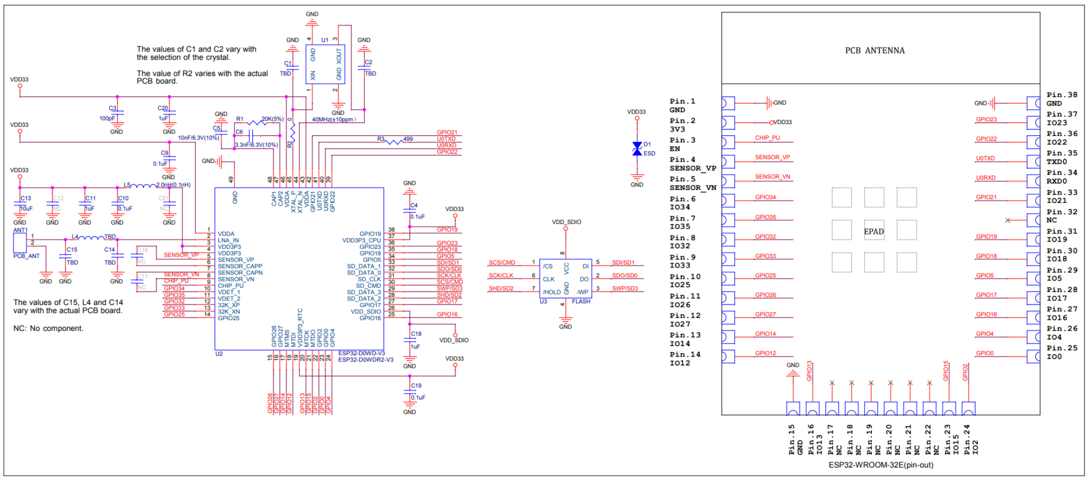


Figure 79: ESP32 WROOM 32E module Schematics ([57] page 22)

A.9 Monarch 2 circuit assembly

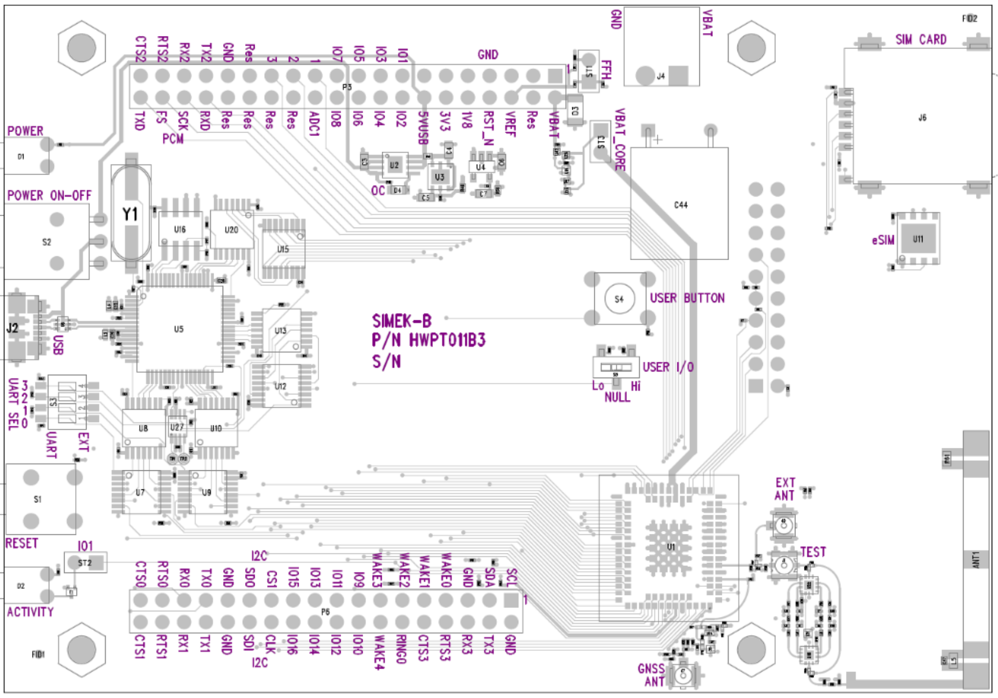


Figure 80: Monarch 2 top assembly (B.3 page 16)

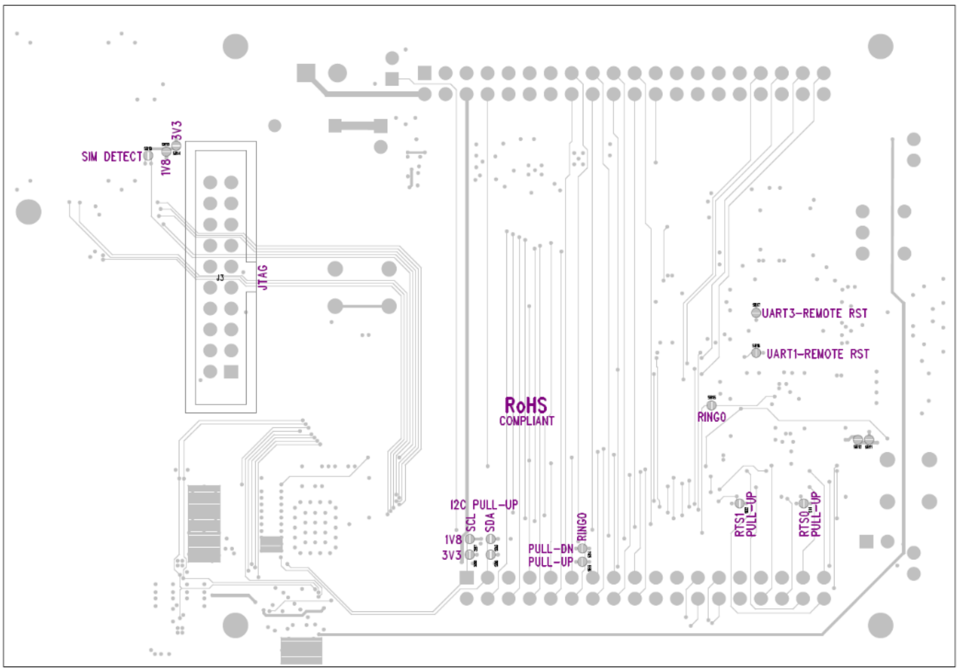


Figure 81: Monarch 2 bottom assembly (B.3 page 17)

A.10 Monarch 2 schematics

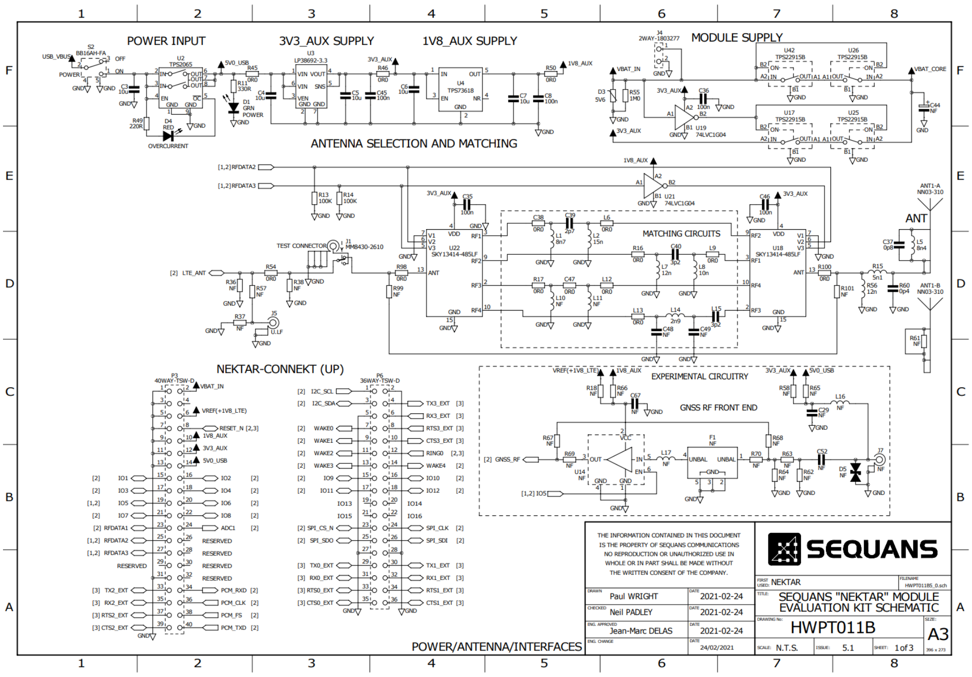

POWER INPUT
3V3_AUX SUPPLY
1V8_AUX SUPPLY
MODULE SUPPLY
ANTENNA SELECTION AND MATCHING
MATCHING CIRCUITS
TEST CONNECTOR
NEKTAR-CONNEKT (UP)
EXPERIMENTAL CIRCUITRY
GNSS RF FRONT END
ANT
SEQUANS
THE INFORMATION CONTAINED IN THIS DOCUMENT IS THE PROPERTY OF SEQUANS COMMUNICATIONS NO REPRODUCTION OR IN PART SHALL BE MADE WITHOUT THE WRITTEN CONSENT OF THE COMPANY.
Paul WRIGHT 2021-02-24
Neil PADLEY 2021-02-24
Jean-Marc DELAS 2021-02-24
24/02/2021
NEKTAR
SEQUANS "NEKTAR" MODULE EVALUATION KIT SCHEMATIC
HWPT011B
A3
N.T.S.
5.1
1 of 3
POWER/ANTENNA/INTERFACES

Figure 82: Monarch 2 evaluation kit schematics part 1

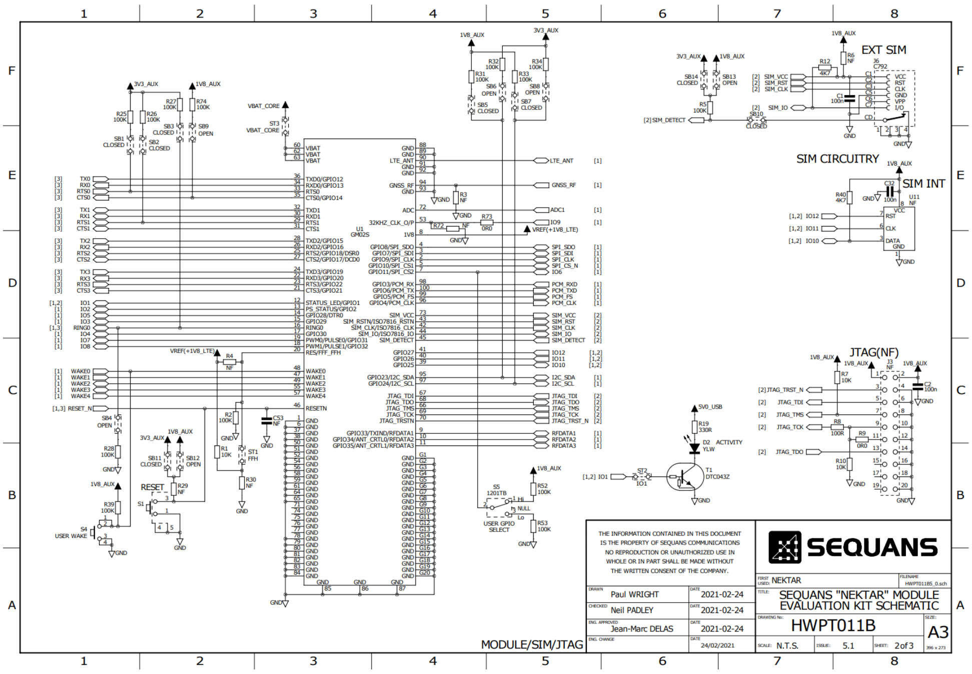


Figure 83: Monarch 2 evaluation kit schematics part 2

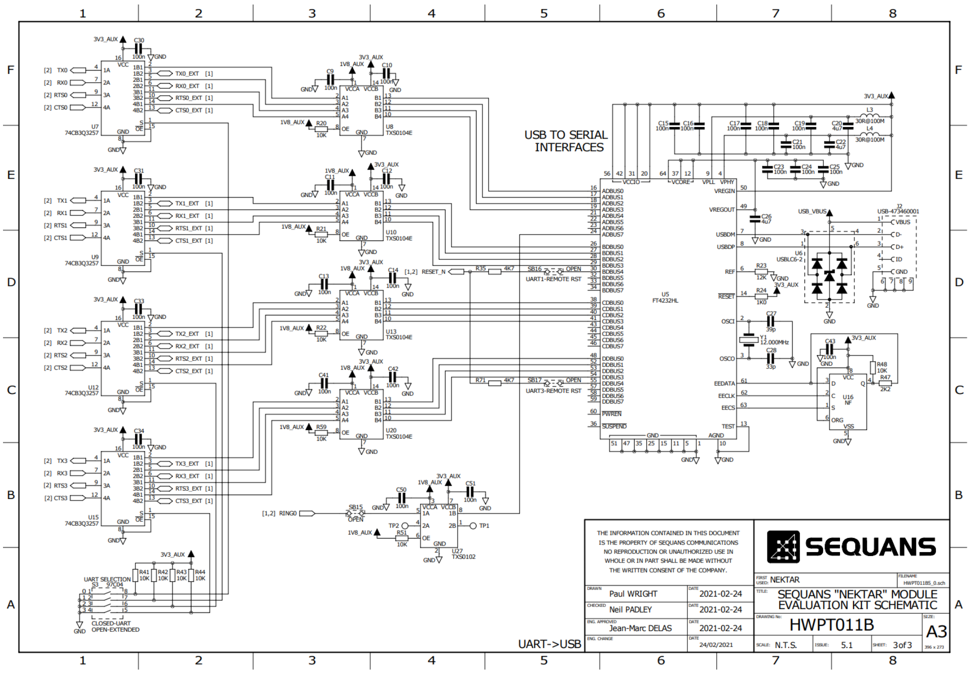


Figure 84: Monarch 2 evaluation kit schematics part 3

A.11 Serial Monitor upon system power-up

When powering up the system, everything in the setup section in the code is executed. Including AT command functions and Wi-Fi configuration. Corresponding actions and responses are printed in Serial Monitor, as depicted below. The last line also depicts the ESP32 going into deep sleep.

```
20:55:30.699 -> Beginning UART Monarch 2 init!
20:55:49.589 ->
20:55:49.589 -> Command:
20:55:49.589 -> AT
20:55:49.589 -> Response:
20:55:49.589 -> OK
20:55:49.589 ->
20:55:49.589 ->
20:55:49.589 ->
20:55:51.586 -> Command:
20:55:51.586 -> AT+CFUN=5
20:55:53.072 -> Response:
20:55:53.072 -> OK
20:55:53.072 ->
20:55:53.072 ->
20:55:53.072 ->
20:55:55.057 -> Command:
20:55:55.057 -> AT+SQNIPSCFG=2,100
20:55:55.103 -> Response:
20:55:55.103 -> +CEREG: 0
20:55:55.103 ->
20:55:55.103 -> OK
20:55:55.103 ->
20:55:55.103 ->
20:55:55.103 ->
20:55:57.110 -> Command:
20:55:57.110 -> AT+SQNHWCFG="uart0","enable","rtscts"
20:55:57.110 -> Response:
20:55:57.110 -> OK
20:55:57.110 ->
20:55:57.110 ->
20:55:57.110 ->
20:55:59.118 -> Command:
20:55:59.118 -> AT+SQNHWCFG="wakeRTS0","enable"
20:55:59.118 -> Response:
20:55:59.118 -> OK
20:55:59.118 ->
20:55:59.118 ->
20:55:59.118 ->
```

```
20:56:01.144 -> Command:
20:56:01.144 -> AT+SQNRICFG=1,3,100
20:56:01.144 -> Response:
20:56:01.144 -> OK
20:56:01.144 ->
20:56:01.144 ->
20:56:01.144 ->
20:56:03.157 -> Command:
20:56:03.157 -> AT^RESET
20:56:03.470 -> Response:
20:56:03.470 -> OK
20:56:03.470 ->
20:56:03.470 ->
20:56:05.469 -> Finished UART init.
20:56:05.469 ->
20:56:05.964 -> Configure Operation Mode!
20:56:05.964 ->
20:56:05.964 -> Command:
20:56:05.964 -> AT+SQNCTM="standard"
20:56:06.556 -> Response:
20:56:06.556 -> +SHUTDOWN
20:56:06.556 ->
20:56:06.556 -> +SYSSTART
20:56:06.556 ->
20:56:06.556 -> OK
20:56:06.556 ->
20:56:06.556 ->
20:56:06.556 ->
20:56:09.561 -> Command:
20:56:09.561 -> AT+SQNCTM?
20:56:09.608 -> Response:
20:56:09.608 -> +SQNCTM: standard
20:56:09.608 ->
20:56:09.608 -> OK
20:56:09.608 ->
20:56:09.608 ->
20:56:09.608 ->
20:56:12.603 -> Command:
20:56:12.603 -> AT+SQNBANDSEL?
20:56:12.638 -> Response:
20:56:12.638 -> +SQNBANDSEL: 0,3gpp-conformance,""
20:56:12.638 -> +SQNBANDSEL: 0,att,"2,4,12"
20:56:12.638 -> +SQNBANDSEL: 0,docomo,"1,19"
20:56:12.638 -> +SQNBANDSEL: 0,kddi,"18,26"
20:56:12.638 -> +SQNBANDSEL: 0,standard,"3,8,20"
```

```
20:56:12.638 -> +SQNBANDSEL: 0,tmo,"2,4,5,12,66"
20:56:12.679 -> +SQNBANDSEL: 0,verizon-no-roaming,"4,13"
20:56:12.679 -> +SQNBANDSEL: 0,verizon,"13,4,5,12,17,20"
20:56:12.679 ->
20:56:12.679 -> OK
20:56:12.679 ->
20:56:12.679 ->
20:56:12.679 ->
20:56:15.676 -> Command:
20:56:15.676 -> AT+SQNBANDSEL=0,"standard","3,8,20"
20:56:15.723 -> Response:
20:56:15.723 -> +SQNBANDSEL: 0,standard,"3,8,20"
20:56:15.723 ->
20:56:15.723 -> OK
20:56:15.723 ->
20:56:15.723 ->
20:56:15.723 ->
20:56:18.719 -> Command:
20:56:18.719 -> AT+SQNEARFCNSEL?
20:56:18.798 -> Response:
20:56:18.798 -> +SQNEARFCNSEL: 0,3gpp-conformance,""
20:56:18.798 -> +SQNEARFCNSEL: 0,att,""
20:56:18.798 -> +SQNEARFCNSEL: 0,docomo,""
20:56:18.798 -> +SQNEARFCNSEL: 0,kddi,""
20:56:18.798 -> +SQNEARFCNSEL: 0,standard,""
20:56:18.798 -> +SQNEARFCNSEL: 0,tmo,""
20:56:18.798 -> +SQNEARFCNSEL: 0,verizon-no-roaming,""
20:56:18.798 -> +SQNEARFCNSEL: 0,verizon,""
20:56:18.798 ->
20:56:18.798 -> OK
20:56:18.798 ->
20:56:18.798 ->
20:56:18.798 ->
20:56:21.829 -> Command:
20:56:21.829 -> AT+SQNEARFCNSEL=0,"standard",""
20:56:22.203 -> Response:
20:56:22.203 -> ERROR
20:56:22.203 ->
20:56:22.203 ->
20:56:22.203 ->
20:56:25.198 -> Command:
20:56:25.198 -> AT+SQNEARFCNSEL?
20:56:25.276 -> Response:
20:56:25.276 -> +SQNEARFCNSEL: 0,3gpp-conformance,""
20:56:25.276 -> +SQNEARFCNSEL: 0,att,""
```

```
20:56:25.276 -> +SQNEARFCNSEL: 0,docomo,""
20:56:25.276 -> +SQNEARFCNSEL: 0,kddi,""
20:56:25.276 -> +SQNEARFCNSEL: 0,standard,""
20:56:25.276 -> +SQNEARFCNSEL: 0,tmo,""
20:56:25.276 -> +SQNEARFCNSEL: 0,verizon-no-roaming,""
20:56:25.276 -> +SQNEARFCNSEL: 0,verizon,""
20:56:25.276 ->
20:56:25.276 -> OK
20:56:25.276 ->
20:56:25.276 ->
20:56:25.276 ->
20:56:28.274 -> Command:
20:56:28.274 -> AT+SQNCTM=?
20:56:28.274 -> Response:
20:56:28.274 -> +SQNCTM: ("standard", "3gpp-conformance", "verizon",
"verizon-no-roaming", "att", "docomo", "kddi", "tmo")
20:56:28.320 ->
20:56:28.320 -> OK
20:56:28.320 ->
20:56:28.320 ->
20:56:28.320 ->
20:56:31.316 -> Command:
20:56:31.316 -> AT+SQNCTM="standard"
20:56:31.362 -> Response:
20:56:31.362 -> OK
20:56:31.362 ->
20:56:31.362 ->
20:56:31.362 ->
20:56:34.359 -> Command:
20:56:34.359 -> AT+SQNCTM=?
20:56:34.359 -> Response:
20:56:34.359 -> +SQNCTM: ("standard", "3gpp-conformance", "verizon",
"verizon-no-roaming", "att", "docomo", "kddi", "tmo")
20:56:34.359 ->
20:56:34.359 -> OK
20:56:34.359 ->
20:56:34.359 ->
20:56:37.369 -> Finished configuring operation mode.
20:56:37.369 ->
20:56:37.868 -> Connecting to NB-IoT network!
20:56:37.868 ->
20:56:37.868 -> Command:
20:56:37.868 -> AT+CFUN=1
20:56:37.868 -> Response:
20:56:37.868 -> OK
```

```
20:56:37.868 ->
20:56:37.868 ->
20:56:37.868 ->
20:56:41.885 -> Command:
20:56:41.885 -> AT+CREG?
20:56:41.885 -> Response:
20:56:41.885 -> +CEREG: 2
20:56:41.885 ->
20:56:41.885 -> +CEREG: 1,"816B","030D6901",7,,,"00001000","10000110"
20:56:41.885 ->
20:56:41.885 -> OK
20:56:41.885 ->
20:56:41.885 ->
20:56:41.885 ->
20:56:45.869 -> Command:
20:56:45.869 -> AT+CGPADDR
20:56:45.869 -> Response:
20:56:45.869 -> +CGPADDR: 1,"10.135.16.20","42.2.33.33.2.71.100.187.
0.0.0.100.66.176.73.1"
20:56:45.869 ->
20:56:45.869 -> OK
20:56:45.869 ->
20:56:45.869 ->
20:56:49.887 -> Connected to Telenor NB-IoT network.
20:56:49.887 ->
20:56:50.400 -> Initializing Power Saveing Mode!
20:56:50.400 ->
20:56:50.400 -> Command:
20:56:50.400 -> AT+SQNEDRX=0
20:56:50.447 -> Response:
20:56:50.447 -> OK
20:56:50.447 ->
20:56:50.447 ->
20:56:50.447 ->
20:56:50.525 -> Command:
20:56:50.525 -> AT+CPSMS=1,,,"10000110","00001000"
20:56:50.604 -> Response:
20:56:50.604 -> OK
20:56:50.604 ->
20:56:50.604 ->
20:56:50.604 ->
20:56:50.697 -> Command:
20:56:50.697 -> AT+CEREG=4
20:56:50.743 -> Response:
20:56:50.743 -> OK
```

```
20:56:50.743 ->
20:56:50.743 ->
20:56:50.743 ->
20:56:50.823 -> Command:
20:56:50.823 -> AT+CFUN=1
20:56:50.823 -> Response:
20:56:50.823 -> OK
20:56:50.823 ->
20:56:50.823 ->
20:56:51.928 -> RTS0 state: 0
20:56:51.928 -> Finished Power Saveing Mode initialization.
20:56:51.928 ->
20:56:53.550 -> Connecting to WiFi ...
20:56:53.550 -> 192.168.43.205
20:56:53.550 ->
20:57:53.549 -> Zzz
```

A.12 Serial Monitor NB-IoT button pushed

```
21:02:47.354 -> NB-IoT button pushed!
21:02:47.354 -> Waking up modem
21:02:47.354 -> RTS0 state: 1
21:02:47.447 -> CTS is clear
21:02:47.447 -> Checking if buffer is ready for data and AT commands
21:02:47.447 -> AT+CFUN?
21:02:47.447 -> Response:
21:02:47.447 -> +CFUN: 1
21:02:47.447 ->
21:02:47.447 -> OK
21:02:47.447 ->
21:02:47.526 -> Ready for PING test
21:02:47.526 -> Command:
21:02:47.526 -> AT+PING="www.sequans.com"
21:02:47.526 -> Response:
21:02:47.572 -> +PING: 1,209.126.25.3,350,50
21:02:47.572 -> +PING: 2,209.126.25.3,200,50
21:02:47.572 -> +PING: 3,209.126.25.3,200,50
21:02:47.572 -> +PING: 4,209.126.25.3,200,50
21:02:47.572 ->
21:02:47.572 -> OK
21:02:47.572 ->
21:02:47.572 ->
21:02:47.572 ->
21:02:47.650 -> RTS0 state: 0
```

A.13 Type function name in Serial Monitor

Function name typed was "uartInitMonarch" in Serial Monitor input field, below is what was printed out.

```
21:06:14.961 -> Beginning UART Monarch 2 init!
21:06:14.961 ->
21:06:14.961 -> Command:
21:06:14.962 -> AT
21:06:14.962 -> Response:
21:06:14.962 -> +CFUN: 1
21:06:14.962 ->
21:06:14.962 -> OK
21:06:14.962 ->
21:06:14.962 ->
21:06:14.962 ->
21:06:16.942 -> Command:
21:06:16.942 -> AT+CFUN=5
21:06:16.989 -> Response:
21:06:16.989 -> +PING: 1,209.126.25.3,210,50
21:06:16.989 -> +PING: 2,209.126.25.3,200,50
21:06:16.989 -> +PING: 3,209.126.25.3,200,50
21:06:16.989 -> +PING: 4,209.126.25.3,200,50
21:06:16.990 ->
21:06:16.990 -> OK
21:06:16.990 ->
21:06:16.990 ->
21:06:16.990 ->
21:06:18.993 -> Command:
21:06:18.993 -> AT+SQNIPSCFG=2,100
21:06:18.993 -> Response:
21:06:18.993 -> OK
21:06:18.993 ->
21:06:18.993 ->
21:06:18.993 ->
21:06:21.000 -> Command:
21:06:21.000 -> AT+SQNHWCFG="uart0","enable","rtscts"
21:06:21.000 -> Response:
21:06:21.000 -> OK
21:06:21.000 ->
21:06:21.000 ->
21:06:21.000 ->
21:06:22.988 -> Command:
21:06:22.988 -> AT+SQNHWCFG="wakeRTS0","enable"
21:06:22.988 -> Response:
21:06:22.988 -> +CEREG: 0
```

```
21:06:22.988 ->
21:06:22.988 -> OK
21:06:22.988 ->
21:06:22.988 ->
21:06:22.988 ->
21:06:25.000 -> Command:
21:06:25.000 -> AT+SQNRICFG=1,3,100
21:06:25.000 -> Response:
21:06:25.000 -> OK
21:06:25.000 ->
21:06:25.000 ->
21:06:25.000 ->
21:06:26.998 -> Command:
21:06:26.998 -> AT^RESET
21:06:26.998 -> Response:
21:06:26.998 -> OK
21:06:26.998 ->
21:06:26.998 ->
21:06:28.998 -> Finished UART init.
```

A.14 Type AT commands in Serial Monitor

Individual AT commands were typed in Serial Monitor input field, that combined initiate UART on Monarch 2 ("uartInitMonarch"). Below is what was printed out.

```
21:09:10.998 -> Command:
21:09:10.998 -> AT
21:09:11.075 -> Response:
21:09:11.075 -> OK
21:09:11.075 ->
21:09:11.075 -> OK
21:09:11.075 ->
21:09:11.075 ->
21:09:11.075 ->
21:09:11.075 ->
21:09:11.075 ->
21:09:11.075 -> OK
21:09:11.075 ->
21:09:18.935 ->
21:09:18.935 -> Command:
21:09:18.935 -> AT
21:09:19.029 -> Response:
21:09:19.029 -> OK
21:09:19.029 ->
```

```
21:09:21.039 ->
21:09:21.039 -> Command:
21:09:21.039 -> AT+CFUN=5
21:09:21.115 -> Response:
21:09:21.115 -> OK
21:09:21.115 ->
21:09:27.965 ->
21:09:27.965 -> Command:
21:09:27.965 -> AT+SQNIPSCFG=2,100
21:09:28.058 -> Response:
21:09:28.058 -> OK
21:09:28.058 ->
21:09:39.759 ->
1:09:57.498 ->
21:09:57.498 -> Command:
21:09:57.498 -> AT+SQNHWCFG="uart0","enable","rtscts"
21:09:57.591 -> Response:
21:09:57.591 -> OK
21:09:57.591 ->
21:10:17.419 ->
21:10:17.420 -> Command:
21:10:17.420 -> AT+SQNHWCFG="wakeRTS0","enable"
21:10:17.545 -> Response:
21:10:17.545 -> OK
21:10:17.545 ->
21:10:24.198 ->
21:10:24.199 -> Command:
21:10:24.199 -> AT+SQNRICFG=1,3,100
21:10:24.277 -> Response:
21:10:24.277 -> OK
21:10:24.277 ->
21:10:33.748 ->
21:10:33.748 -> Command:
21:10:33.748 -> AT^RESET
21:10:33.842 -> Response:
21:10:39.667 -> +SHUTDOWN
21:10:39.667 -> ?
21:10:39.730 -> +SYSSTART
```

B Attachments

The submission of this bachelor thesis includes different documents in addition to the project report. Here are the different documents and folders listed. The main project code is also printed out, to make the report more comprehensibly for readers that do not have access to the complete submission.

B.1 Project documentation

There was a Microsoft Teams set up for this project with a number of folders and files relevant to the project and process. Note that the titles are written in Norwegian, as this was the working language with the supervisors. The documents related to Monarch 2 is also placed as their own attachments to make it easier to refer to them in the text.

Folders and contents:

- Dokumenter (Documents)
  - Bachelor form
  - Different documents about Monarch 2 provided by Sequans Communications, see other attachments below.
- Forprosjekt (Pre-project)
- Maler (Templates)
- Møteraferater (Meeting minutes) - also worked as status reports
- Sluttprodukt (Final product)
  - Project report
  - Poster
  - Final project code of demonstration setup
  - Standalone Narrowband IoT demonstration
  - Standalone Wi-Fi demonstration
  - Latex code of project report

B.2 Monarch 2 datasheet

Name of file: Monarch2-GM02S-DataSheet-Rev.14

B.3 Monarch 2 NEKTAR-B User Manual

File name: Monarch2-NEKTARB-EVK-UsersManual-TruPhone-Rev.2

B.4 Monarch 2 AT Commands Use Cases

File name: Monarch2-LR8.0-ATCommandsUseCases-Rev.6

B.5 Monarch 2 System Integration Guide

File name: Monarch2-SystemIntegrationGuide-Rev.8a

B.6 Monarch 2 Module Integration Guide

File name: NDA - GM02S_ModuleIntegrationGuide_Rev2

B.7 Monarch 2 release Notes R02

File name: LR8.0.5.12 Release Notes_R02

B.8 Monarch 2 Power Consumption Measurements

File name: NDA- Monarch2-PowerConsumptionMeasurementAppNote_Rev3

B.9 Monarch 2 AT Commands Reference Manual

File name: Monarch2-LR8.0-ATCommandsReferenceManual-Rev.9

B.10 Serial Monitor as terminal window

Basic functionality. Enable Serial Monitor in Arduino IDE to be a functioning terminal where the ESP32 controls the Monarch 2.
File name: type_at_esp32_monarch2.ino

B.11 Standalone Narrowband IoT demonstration

Standalone NB-IoT demonstration part from final project.
File name: NB_demo_cot_devkit_iot.ino

B.12 Standalone Wi-Fi demonstration

Standalone Wi-Fi demonstration part from final project.
File name: WIFI_demo_cot_devkit_iot.ino

B.13 Final project code: Wi-Fi and NB-IoT demo

The main project code for CoT DevKit IoT proof-of-concept with demonstration setup with Wi-Fi and NB-IoT.
File name: Final_complete_code-WIFI_and_NB_demo_cot_devkit_iot.ino

```
/*
  * CoT DevKit IoT --> NB-IoT and Wi-Fi demo
  * Filename: Final_complete_code-WIFI_and_NB_demo_cot_devkit_iot.ino

  * NB-IoT:
    - ESP32 controls modem Monarch 2 through UART
    - Type in AT commands or function name in Serial Monitor
    - Physical button pushed sends PING test to network

  * Wi-Fi:
    - ESP32 acts as access point and web server
    - Physical button pushed and state updated on webpage
    - Physical button wake ESP32 (also effects Monarch 2)
*/


// Libraries
// ESP32 library already included in Arduino IDE (check if you have esp_sleep.h)
#include <HardwareSerial.h>     // Serial/UART communication library
#include <WiFi.h>               // Library for Wi-Fi with ESP
#include <AsyncTCP.h>           // Asynchronous TCP library for ESP
#include <ESPAsyncWebServer.h>  // Asynchronous WebServer library for ESP
/* Self-made library for communication between Monarch 2 and ESP32
   Defining UART configurations, ports and hardware flow control pins
   Operational command functions to be included in program or typed manually in Serial Monitor */
#include "CoT_Monarch.h"

#define BAUD_RATE 115200

const int esp32_wake_BUTTON = 33;             // Wake up button connected to this pin on ESP32
unsigned long currentTime = 0;                // Keep track of last time buttons pressed
const unsigned long lightSleepTime = 60000;   // Wait this long after Wi-Fi button inactive to go into lgiht sleep

// NB-IoT variables
const int NB_BUTTON = 4;  // NB button connected to this pin on ESP32
bool nb_buttonState = false;
bool nb_lastButtonState = false;
unsigned long nb_lastButtonPressTime = 0;

// Wi-Fi variables
const char* SSID = "Xperia␣XA_22c1";   // Wi-Fi network name
const char* PASSWORD = "annikerbest";  // Password
const char* INPUT_PARAM = "state";     // Save parameter for webpage
const int wifi_BUTTON = 0;             // Wi-Fi button connected to this pin on ESP32
const int wifi_output_point = 2;       // Test pin - output
int wifi_outputState = LOW;
int wifi_buttonState;
int wifi_lastButtonState = LOW;
unsigned long wifi_lastButtonPressTime = 0;
unsigned long lastDebounceTime = 0;  // For webpage to keep track
unsigned long debounceDelay = 50;

// Enable web server on port 80 (HTTP)
AsyncWebServer server(80);

// Webpage HTML code - inspired by https://randomnerdtutorials.com/esp32-web-server-arduino-ide/
const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE␣HTML><html>
<head>
␣␣<title>CoT␣DevKit␣IoT␣-␣WiFi␣demo</title>
␣␣<meta␣name="viewport"␣content="width=device-width, initial-scale=1">
␣␣<style>
␣␣␣␣html␣{font-family:␣Arial;␣display:␣inline-block;␣text-align:␣center;}
␣␣␣␣h2␣{font-size:␣2.5rem;}
␣␣␣␣p␣{font-size:␣2.5rem;}
␣␣␣␣body␣{max-width:␣600px;␣margin:0px␣auto;␣padding-bottom:␣25px;}
␣␣␣␣.switch␣{position:␣relative;␣display:␣inline-block;␣width:␣180px;␣height:␣100px}
␣␣␣␣.switch␣input␣{display:␣none}
␣␣␣␣.slider␣{position:␣absolute;␣top:␣0;␣left:␣0;␣right:␣0;␣bottom:␣0;␣background-color:␣#
    D3D3D3;␣border-radius:␣40px}
␣␣␣␣.slider:before␣{position:␣absolute;␣content:␣"";␣height:␣80px;␣width:␣80px;␣left:␣10px;␣
    bottom:␣10px;␣background-color:␣#FFF;␣-webkit-transition:␣.4s;␣transition:␣.4s;␣border-
    radius:␣70px}
␣␣␣␣input:checked␣+␣.slider␣{background-color:␣#68a0b4}
␣␣␣␣input:checked␣+␣.slider:before␣{-webkit-transform:␣translateX(80px);␣-ms-transform:␣
    translateX(80px);␣transform:␣translateX(80px)}
␣␣</style>
</head>
<body>
␣␣<h2>CoT␣DevKit␣IoT␣-␣WiFi␣demo</h2>
␣␣%BUTTONPLACEHOLDER%
<script>function␣toggleCheckbox(element)␣{
␣␣var␣xhr␣=␣new␣XMLHttpRequest();
␣␣if(element.checked){␣xhr.open("GET",␣"/update?state=1",␣true);␣}
␣␣else␣{␣xhr.open("GET",␣"/update?state=0",␣true);␣}
␣␣xhr.send();
}
setInterval(function␣(␣)␣{
␣␣var␣xhttp␣=␣new␣XMLHttpRequest();
␣␣xhttp.onreadystatechange␣=␣function()␣{
␣␣␣␣if␣(this.readyState␣==␣4␣&&␣this.status␣==␣200)␣{
␣␣␣␣␣␣var␣inputChecked;
␣␣␣␣␣␣var␣outputStateM;
␣␣␣␣␣␣if(␣this.responseText␣==␣1){
␣␣␣␣␣␣␣␣inputChecked␣=␣true;
␣␣␣␣␣␣␣␣outputStateM␣=␣"On";
␣␣␣␣␣␣}
␣␣␣␣␣␣else␣{
␣␣␣␣␣␣␣␣inputChecked␣=␣false;
␣␣␣␣␣␣␣␣outputStateM␣=␣"Off";
␣␣␣␣␣␣}
␣␣␣␣␣␣document.getElementById("wifi_output_point").checked␣=␣inputChecked;
␣␣␣␣␣␣document.getElementById("wifi_outputState").innerHTML␣=␣outputStateM;
␣␣␣␣}
␣␣};
␣␣xhttp.open("GET",␣"/state",␣true);
␣␣xhttp.send();
},␣1000␣)␣;
</script>
</body>
</html>
)rawliteral";

// Replace placeholder with actual button state on webpage - inspired by same source
String processor(const String& var) {
  if (var == "BUTTONPLACEHOLDER") {
    String buttons = "";
    String wifi_outputStateValue = wifi_check_outputState();
    buttons += "<h4><span␣id=\"wifi_check_outputState\"></span></h4><label␣class=\"switch\"><
        input␣type=\"checkbox\"␣onchange=\"toggleCheckbox(this)\"␣id=\"wifi_output_point\"␣" +
         wifi_outputStateValue + "><span␣class=\"slider\"></span></label>";
    return buttons;
  }
  return String();
}

```

```
// Check state on output point to place on webpage - inspired by same source
String wifi_check_outputState() {
  if (digitalRead(wifi_output_point)) {
    return "checked";
  } else {
    return "";
  }
  return "";
}


//======================== VOID SETUP ========================//

void setup() {
  Serial.begin(BAUD_RATE);

  // Enable physical button as wake up source after ESP32 has entered light sleep mode
  pinMode(esp32_wake_BUTTON, INPUT_PULLUP);
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_33, 0);

  //------------------- NB-IoT setup -------------------//

  // Begin UART communication
  esp32Serial.begin(BAUD_RATE);    // Between ESP32 and PC
  monarchSerial.begin(BAUD_RATE);  // Between ESP32 and Monarch 2

  Serial.setRxBufferSize(0);  // Disable software serial buffer

  pinMode(NB_BUTTON, INPUT_PULLUP);  // Configure the button pin as input with internal pull-up resistor

  // Configure hardware flow control pins
  pinMode(RTS_pin, OUTPUT);
  pinMode(CTS_pin, INPUT);
  pinMode(RING_pin, INPUT);

  /*
    * Functions that can be executed on startup
    * Placed in order of necessary execution
    * Required: uartInitMonarch, configureOperationMode, connectNetwork
    * Rest is optional, choose between PSM and eDRX
  */
  uartInitMonarch();  // Configure UART between ESP32 and Monarch
  // selectSIM();               // Select SIM slot
  // powerSIM();                // Power correct SIM slot
  configureOperationMode();  // Configure Monarch Operation Mode
  connectNetwork();          // Connect Monarch to NB-IoT Network
  initPSM();                 // Initialize Power Saving Mode
  // init_eDRX();               // Initialize Extended Discontinuous Reception (eDRX)
  // scanNetwork();             // Informal Network Scan (can be done before PSM and eDRX)

  //------------------- Wi-Fi setup -------------------//

  // Configure Wi-Fi button and test point
  pinMode(wifi_BUTTON, INPUT_PULLUP);
  pinMode(wifi_output_point, OUTPUT);
  digitalWrite(wifi_output_point, LOW);

  // Connect to Wi-Fi
  WiFi.begin(SSID, PASSWORD);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting␣to␣WiFi␣...");
  }

  // Print ESP32's IP address (URL of webpage)
  Serial.println(WiFi.localIP());
  Serial.println();

  // Route for root/webpage
  server.on("/", HTTP_GET, [](AsyncWebServerRequest* request) {
    request->send_P(200, "text/html", index_html, processor);
  });

  // GET request to <ESP_IP>/update?state=<inputMessage>
  server.on("/update", HTTP_GET, [](AsyncWebServerRequest* request) {
    String inputMessage;
    String inputParam;

    // GET value on <ESP_IP>/update?state=<inputMessage>
    if (request->hasParam(INPUT_PARAM)) {
      inputMessage = request->getParam(INPUT_PARAM)->value();
      inputParam = INPUT_PARAM;
      digitalWrite(wifi_output_point, inputMessage.toInt());
      wifi_outputState = !wifi_outputState;
    } else {
      inputMessage = "No␣message␣sent";
      inputParam = "none";
    }

    Serial.println(inputMessage);
    request->send(200, "text/plain", "OK");
  });

  // GET request to <ESP_IP>/state
  server.on("/state", HTTP_GET, [](AsyncWebServerRequest* request) {
    request->send(200, "text/plain", String(digitalRead(wifi_output_point)).c_str());
  });

  // Start server
  server.begin();
}


//======================== VOID LOOP ========================//

void loop() {
  currentTime = millis();  // Keep time configure buttons, remove noise and define sleep time

  //----------------- NB-IoT button (send PING) -----------------//

  nb_buttonState = digitalRead(NB_BUTTON);  // Read the current button state

  // Check if the button state has changed (button pushed)
  if (nb_buttonState != nb_lastButtonState && nb_buttonState == LOW) {
    Serial.println("NB-IoT␣button␣pushed!");

    // Toggle RTS0 to wake modem if in power save mode (PSM or eDRX)
    // RTS0 is configured as wake source
    Serial.println("Waking␣up␣modem");
    digitalWrite(RTS_pin, HIGH);
    int rts_state = digitalRead(RTS_pin);
    Serial.print("RTS0␣state:␣");
    Serial.println(rts_state);
    delay(100);

    // Check if CTS pin is low (indicating modem is ready to receive data and AT commands)
    if (digitalRead(CTS_pin) == LOW) {
      Serial.println("CTS␣is␣clear");

      String checkReady = "AT+CFUN?";
      monarchSerial.println(checkReady);
      Serial.println("Checking␣if␣buffer␣is␣ready␣for␣data␣and␣AT␣commands");
      Serial.println(checkReady);

      // Handle response from Monarch and send PING test to www.sequans.com to see that it is attached to NB-IoT network
      buttonNB_handleResponse_sendPING();

    } else {
      Serial.println("CTS␣not␣clear␣and␣Monarch␣2␣not␣ready");
    }
    delay(100);

    // Toggle RTS pin after data transmission is complete
    digitalWrite(RTS_pin, LOW);
    Serial.print("RTS0␣state:");
    Serial.println(rts_state);

    Serial.println();
    nb_lastButtonPressTime = millis();
    delay(100);
  }

  nb_lastButtonState = nb_buttonState;  // Update button state


  //-------- Type AT command or Function name in Serial Monitor --------//

  // Monitoring uart line
  if (esp32Serial.available()) {

    // Read input typed in Serial Monitor
    String input = esp32Serial.readStringUntil('\n');
    input.trim();  // Remove leading/trailing whitespaces

    // Check if the input is an AT command
    if (input.startsWith("AT")) {

      // Send the AT command to Monarch
      monarchSerial.println(input);  // Forward data
      Serial.println();
      Serial.println("Command:␣");
      Serial.println(input);  // Echo to display what typed

      delay(100);

      // Read and print the response from Monarch
      Serial.print("Response:␣");
      while (monarchSerial.available()) {
        char c = monarchSerial.read();
        esp32Serial.write(c);  // Forward data
      }

      Serial.println();
    }

    /*
      * Else a function name was typed.
      * Function names available (see CoT_Monarch.h library):
       - uartInitMonarch         // Configure UART between ESP32 and Monarch
       - selectSIM               // Select SIM slot
       - powerSIM                // Power correct SIM slot
       - configureOperationMode  // Configure Monarch Operation Mode
       - connectNetwork          // Connect Monarch to NB-IoT Network
       - initPSM                 // Initialize Power Saving Mode
       - init_eDRX               // Initialize Extended Discontinuous Reception (eDRX)
       - scanNetwork             // Informal Network Scan (can be done before PSM and eDRX)
    */
    else {
      executeFunction(input);
    }
  }


  //------------ Wi-Fi button (change state on webpage) ------------//

  int wifi_actual_buttonState = digitalRead(wifi_BUTTON);  // Read the current button state

  // If button pushed, save time
  if (wifi_actual_buttonState != wifi_lastButtonState) {
    wifi_lastButtonPressTime = millis();
  }
  // If button pushed, change state on webpage
  if ((currentTime - wifi_lastButtonPressTime) > debounceDelay) {
    if (wifi_actual_buttonState != wifi_buttonState) {
      wifi_buttonState = wifi_actual_buttonState;
      if (wifi_buttonState == HIGH) {
        wifi_outputState = !wifi_outputState;
      }
    }
  }

  digitalWrite(wifi_output_point, wifi_outputState);  // Change state on output test point
  wifi_lastButtonState = wifi_actual_buttonState;      // Update button state


  //---------- ESP32 enter Light Sleep Mode after inactivity ----------//

  // Entering Light Sleep Mode after both buttons inactive for defined value of lightSleepTime
  if ((currentTime - nb_lastButtonPressTime >= lightSleepTime) && (currentTime - wifi_lastButtonPressTime >= lightSleepTime)) {
    Serial.println("Zzz");
    esp_light_sleep_start();                 // Enter light sleep mode, function from esp_sleep.h library in ESP32 add-on
    nb_lastButtonPressTime = currentTime;    // Reset last button pressed times
    wifi_lastButtonPressTime = currentTime;  // Reset last button pressed times
  }
}

//===========================================================//
```

B.14 CoT_Monarch library

B.14.1 CoT_Monarch library header file

```
/*
  * CoT_Monarch.h is a library designed for CoT DevKit IoT
  * It enables communication between Monarch 2 (DCE) and ESP32 (DTE)
  * Stores useful functions
*/

#ifndef COT_MONARCH_H
#define COT_MONARCH_H

#include <Arduino.h>
#include <HardwareSerial.h>

extern HardwareSerial monarchSerial;
extern HardwareSerial esp32Serial;

/*
  * Hardware flow control pins on ESP32 connected to corresponding pins on Monarch 2
  * RX and TX are defined in monarchSerial
*/
const int RTS_pin = 14;   // Request to Send, also wake up source for Monarch 2
const int CTS_pin = 15;   // Clear to Send
const int RING_pin = 32;  // Line that monitors data and URC on UART line

// Type function names in Serial Monitor to execute manually
void executeFunction(String function);

// Process response from Monarch 2 to ESP32
void handleResponse();

// For NB-IoT PING test button -> processes response from Monarch and send PING test to www.sequans.com to see that it is attached to NB-IoT network
void buttonNB_handleResponse_sendPING();

// Initializes UART communication with hardware flow control between Monarch 2 and ESP32
void uartInitMonarch();

// Selecting SIM slot on Monarch 2
void selectSIM();

// Power correct SIM sloton Monarch 2
void powerSIM();

// Configure Monarch 2 Operation Mode
void configureOperationMode();

// Connect Monarch 2 to NB-IoT Network
void connectNetwork();

// Initialize low-power mode: Power Saving Mode
void initPSM();

// Initialize low-power mode: Extended Discontinuous Reception (eDRX)
void init_eDRX();

// Informal Network Scan
void scanNetwork();

#endif
```

B.14.2 CoT_Monarch library CPP file

```
#include "CoT_Monarch.h"

// UART configuration between ESP32 and Monarch 2 uses ESP32 UART2
HardwareSerial monarchSerial(2);
// UART configuration between ESP32 and PC uses ESP32 UART0 (USB cable)
HardwareSerial esp32Serial(0);

//--------- Execute function when typed in Serial Monitor ---------//

void executeFunction(String function) {
  // All function names created will be included here

  // Configure UART bewteen ESP32 and Monarch 2
  if (function == "uartInitMonarch") {
    uartInitMonarch();
    delay(100);
  }

  // Select SIM slot
  if (function == "selectSIM") {
    selectSIM();
    delay(100);
  }

  // Power correct SIM slot
  if (function == "powerSIM") {
    powerSIM();
    delay(100);
  }

  // Configure Monarch Operation Mode
  if (function == "configureOperationMode") {
    configureOperationMode();
    delay(100);
  }

  // Connect Monarch to NB-IoT Network
  if (function == "connectNetwork") {
    connectNetwork();
    delay(100);
  }

  // Initialize Power Save Mode
  if (function == "initPSM") {
    initPSM();
    delay(100);
  }

  // Initialize Extended Discontinuous Reception (eDRX)
  if (function == "init_eDRX") {
    init_eDRX();
    delay(100);
  }

  // Informal Network Scan
  if (function == "scanNetwork") {
    scanNetwork();
    delay(100);
  }
}

//---------------- Response from Monarch ----------------//

void handleResponse() {

  // Read and forward response
  String response;
  bool errorDetected = false;
  bool responseComplete = false;
  unsigned long startTime = millis();

  while (millis() - startTime < 5000) {  // Wait for response
    if (monarchSerial.available()) {
      char c = monarchSerial.read();
      response += c;

      // Check if the response starts with "\r\nOK\r\n" or ends with "\r\nOK\r\n" or ends with
           "\r\nERROR\r\n"
      if (response.startsWith("\r\nOK\r\n") || response.endsWith("\r\nOK\r\n") || response.
          endsWith("\r\nERROR\r\n")) {
        responseComplete = true;
        break;
      }
    }
  }

  if (responseComplete) {
    // Print the response
    Serial.print("Response:␣");
    Serial.println(response);

    // Check if the response contains "ERROR"
    if (response.indexOf("ERROR") != -1) {
      errorDetected = true;  // Stops execution of rest of AT commands
    }
  }

  else {
    Serial.println("No␣complete␣response␣received");
    errorDetected = true;  // Stops execution of rest of AT commands
  }

  Serial.println();
}

//------- Handle Response from Monarch and send test PING ------//

void buttonNB_handleResponse_sendPING() {

  // Read and forward the response
  String response;
  bool errorDetected = false;
  bool responseComplete = false;
  unsigned long startTime = millis();

  while (millis() - startTime < 5000) {  // Wait for response
    if (monarchSerial.available()) {
      char c = monarchSerial.read();
      response += c;

      // Check if the response starts with "\r\nOK\r\n" or ends with "\r\nOK\r\n" or ends with
           "\r\nERROR\r\n"
      if (response.startsWith("\r\nOK\r\n") || response.endsWith("\r\nOK\r\n") || response.
          endsWith("\r\nERROR\r\n")) {
        responseComplete = true;
        break;
      }
    }
  }

  if (responseComplete) {
    // Print the response
    Serial.print("Response:␣");
    Serial.println(response);

    // Check if the response contains "ERROR"
    if (response.indexOf("ERROR") != -1) {
      errorDetected = true;
    }
  }

  else {
    Serial.println("No␣complete␣response␣received");
    errorDetected = true;  // Stops execution of rest of AT commands
  }

  delay(100);

  //If ready to for AT command, do ping check
  if (response.endsWith("\r\nOK\r\n")) {
    Serial.println("Ready␣for␣PING␣test");

    // Send command to the modem
    String pingCommand = "AT+PING=\"www.sequans.com\"";
    monarchSerial.println(pingCommand);
    // Print the command
    Serial.println("Command:␣");
    Serial.println(pingCommand);

    handleResponse();
  }

  Serial.println();
}

//---------- Configure UART between ESP32 and Monarch ---------//

void uartInitMonarch() {

  Serial.println("Beginning␣UART␣Monarch␣2␣init!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    "AT",                                           // Test AT command connection
    "AT+CFUN=5",                                    // Entering manufacturing mode
    "AT+SQNIPSCFG=2,100",                           // UART timeout 100ms
    "AT+SQNHWCFG=\"uart0\",\"enable\",\"rtscts\"",  // Enable hardware flow control on Monarch
         UART0
    "AT+SQNHWCFG=\"wakeRTS0\",\"enable\"",          // Setting RTS0 as wake source
    "AT+SQNRICFG=1,3,100",                          // RING timeout 100ms
    "AT^RESET"
  };

  // Loop through each command and forward to Monarch
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it is the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(2000);
  }

  Serial.println("Finished␣UART␣init.");
  Serial.println();
  delay(500);
}

//---------- Select SIM slot ---------//

void selectSIM() {

  Serial.println("Selecting␣SIM␣card␣on␣Monarch!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    // Select SIM slot
    "AT+CSUS=?",  // Check number of SIM slots
    "AT+CSUS?",   // Checking that SIM slot 0 is selected
    "AT+CFUN=0",
    "AT+CSUS=0",  // Selecting SIM slot 0
    "AT+CSUS?",
    "AT+CFUN=4",  // Set to ariplaine mode
    "AT+CIMI",
    "AT+CSUS?",
    "AT^RESET",
    "AT+CSUS?"
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(3000);  // Delay between commands to allow for stability
  }

  Serial.println("Finished␣selecting␣SIM.");
  Serial.println();
  delay(500);
}

//---------- Power correct SIM slot ---------//

void powerSIM() {

  Serial.println("Power␣SIM␣card␣on␣Monarch!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    // Power SIM slot
    "AT+CFUN=1",  // Enter full functionality mode
    "AT+CPIN?"    // Check if SIM is ready
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(2000);  // Delay between commands to allow for stability
  }

  Serial.println("Finished␣powering␣SIM.");
  Serial.println();
  delay(500);
}

//---------- Configure Monarch Operation Mode ---------//

void configureOperationMode() {

  Serial.println("Configure␣Operation␣Mode!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    // Configure Operation Mode of the modem

    // Check current mode
    "AT+SQNCTM=\"standard\"",
    "AT+SQNCTM?",

    // Limit number of bands
    "AT+SQNBANDSEL?",
    "AT+SQNBANDSEL=0,\"standard\",\"3,8,20\"",

    // Defined preffered EARFNC (E-UTRA Absolute Radio Frequency Channel Number) to be scanned in priority
    "AT+SQNEARFCNSEL?",
    "AT+SQNEARFCNSEL=0,\"standard\",\"\"␣",  // Enter an empty list to reset the EARFCN list to the default valuees
    "AT+SQNEARFCNSEL?",

    // Select operation mode
    "AT+SQNCTM=?",
    "AT+SQNCTM=\"standard\"",
    "AT+SQNCTM=?"
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(3000);  // Delay between commands to allow for stability
  }

  Serial.println("Finished␣configuring␣operation␣mode.");
  Serial.println();
  delay(500);
}

//---------- Connect Monarch to NB-IoT Network ---------//

void connectNetwork() {

  Serial.println("Connecting␣to␣NB-IoT␣network!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    // Connect to the Network and Check Attach is Done
    "AT+CFUN=1",  // Attach to network
    "AT+CREG?",   // Check network's registration status
    "AT+CGPADDR"  // Check IP address
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(4000);  // Delay between commands to allow for stability
  }

  Serial.println("Connected␣to␣Telenor␣NB-IoT␣network.");
  Serial.println();
  delay(500);
}


//---------- Initialize Power Saving Mode ---------//

void initPSM() {

  Serial.println("Initializing␣Power␣Saveing␣Mode!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    "AT+SQNEDRX=0",  // Disabling eDRX

    // Enable PSM
    // "10100011": Requested Periodic TAU timer (T3412 Extended), 3 min
    // "00100001": Requested Active timer (T3324), 1 min
    "AT+CPSMS=1,,,\"10000110\",\"00001000\"",

    "AT+CEREG=4",
    "AT+CFUN=1"
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(100);  // Delay between commands to allow for stability
  }

  //Toogle RTS0 LOW and enter sleep mode
  digitalWrite(RTS_pin, LOW);
  delay(1000);
  int real_rts = digitalRead(RTS_pin);
  Serial.print("RTS0␣state:␣0");
  // Serial.println(real_rts);
  Serial.println();

  Serial.println("Finished␣Power␣Saveing␣Mode␣initialization.");
  Serial.println();
  delay(500);
}

//-------- Initialize Extended Discontinuous Reception (eDRX) -------//

void init_eDRX() {

  Serial.println("Initializing␣Extended␣Discontinuous␣Reception␣(eDRX)!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    "AT+CPSMS=0",  // Disabling PSM

    /*  Enable eDRX and unsolicited result code
      * Type of access technology, 4 = E-UTRAN (WB-S1 mode) and 5 = E-UTRAN (NB-S1 mode)
      * eDRX cycle 20.48s
      * WB-S1 PTW Value 2.65s and NB-S1 PTW Value 5.12s
    */
    "AT+SQNEDRX=2,5,\"0010\",\"0001\"",
    "AT+SQNEDRX?",
    "AT+CEDRXS?"
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(100);  // Delay between commands to allow for stability
  }

  // Toogle RTS0 low
  digitalWrite(RTS_pin, LOW);
  delay(1000);
  int real_rts = digitalRead(RTS_pin);
  Serial.print("RTS0␣state:␣");
  Serial.println(real_rts);
  Serial.println();

  Serial.println("Finished␣eDRX␣initialization.");
  Serial.println();
  delay(500);
}

//---------- Informal Network Scan ---------//

void scanNetwork() {

  Serial.println("Beginning␣informal␣network␣scan!");
  Serial.println();

  // List of AT commands to send
  String commands[] = {
    "AT+CFUN=0",    // Enter minimum functionality
    "AT+SQNINS=0",  // Scan all bands
    "AT+SQNMONI=9"  // Report information
  };

  // Loop through each command
  for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {
    // Send command to the modem
    String currentCommand = commands[i];
    monarchSerial.println(currentCommand);

    // Print the command
    Serial.println("Command:␣");
    Serial.println(currentCommand);

    handleResponse();

    // Skip printing the extra line if it's the last command
    if (i < sizeof(commands) / sizeof(commands[0]) - 1) {
      Serial.println();
    }

    delay(4000);  // Delay between commands to allow for stability
  }

  Serial.println("Finished␣informal␣network␣scan.");
  Serial.println();
  delay(500);
}
```

B.15 Poster

Poster for the bachelor project.

CoT DevKit IoT: Internet of Things hardware with Wi-Fi and narrowband IoT capabilities specialised for use in the Norwegian educational system

NTNU | Norwegian University of Science and Technology

Annik Riise
Department of Electronic Systems NTNU

Company of Things

Abstract

Proof-of-concept of a development board intended to teach university and upper secondary school students about Internet of Things (IoT). Enabling them to create their own projects, specifically with Wi-Fi and narrowband IoT (NB-IoT).

ESP32 has an integrated Wi-Fi modem and is the MCU that controls the whole system. Monarch 2 is the NB-IoT modem and is configured with AT commands. Everything is accomplished within Arduino IDE with its Serial Monitor working as the terminal.

Demonstration setup

This project consists of the CoT DevKit IoT with a demonstration system that has the functionalities below.

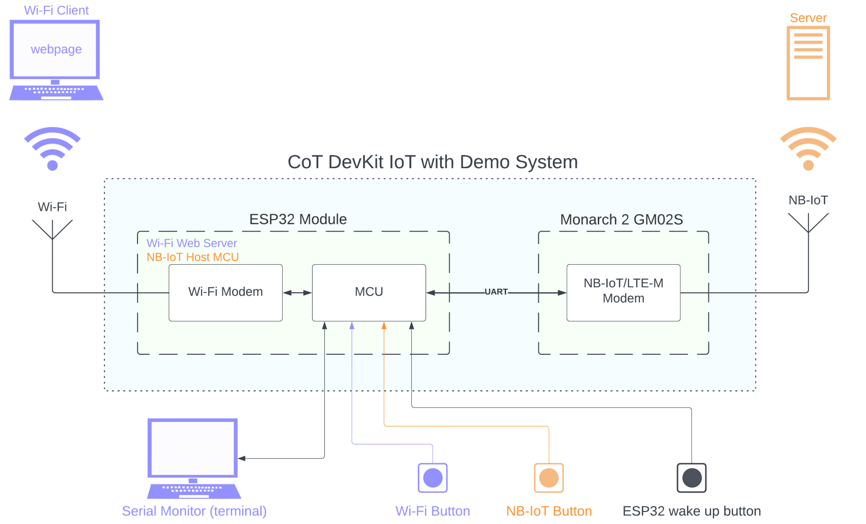
Sending AT commands:
Single AT commands and function names (predefined combination of commands) can manually be typed in Serial Monitor. The function names can also be automatically executed by placing them in the program code.

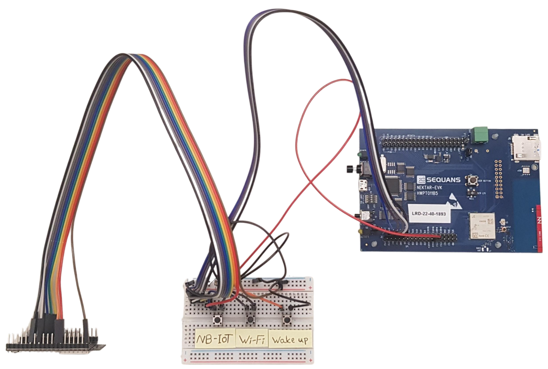
Wi-F button:
ESP32 acts as a web server and access point, hosting a webpage. Pressing the physical Wi-Fi button updates the button state on the webpage.

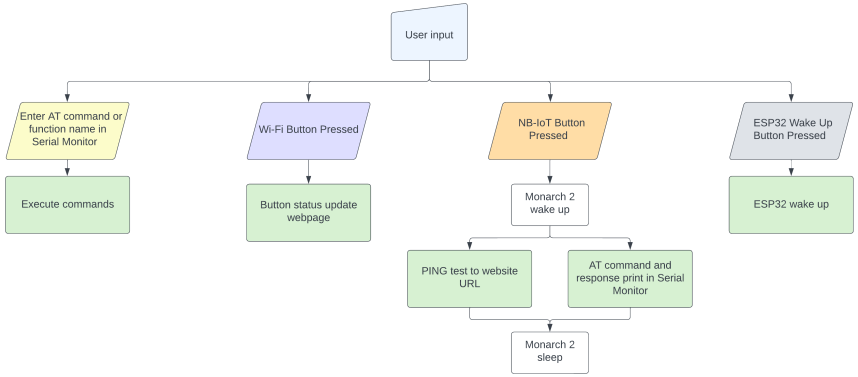
NB-IoT button:
Pressing the NB-IoT button wakes Monarch 2 up from power saving mode, sends a PING connectivity test to a website and goes afterwards back to sleep.

ESP32 wake up button:
Monarch 2 is in constant power saving mode. After 1 min, ESP32 enters sleep mode, seizing all system functionalities – including Monarch 2. ESP32 wake up button is pressed to wake the system up again.

CoT DevKit IoT with Demo System






User input
Enter AT command or function name in Serial Monitor
Execute commands
Wi-Fi Button Pressed
Button status update webpage
NB-IoT Button Pressed
Monarch 2 wake up
PING test to website URL
AT command and response print in Serial Monitor
Monarch 2 sleep
ESP32 Wake Up Button Pressed
ESP32 wake up

Conclusion & Future work

Project resulted in a proof-of-concept that works as intended with the desired specifications. This thesis is the beginning of the bigger design process of the development board. Below are prioritised suggestions for future work:

- More comprehensive hardware flow control with RING line monitoring and perhaps a watchdog.
- Propper power consumption measurements with protective circuit and accurate equipment.
- Further NB-IoT data transmission testing, preferably with the IoT platform Deploii.

Technical specifications

- Main components: ESP32 DevKitC V4 and Monarch 2 GM02S Nektar Evaluation Kit
- Short-range wireless communication for indoor use: Wi-Fi
- Long-range wireless communication for outdoor use: Narrowband IoT (NB-IoT)
- Entry-level microcontroller programming language: Arduino C
- Low-power modes and techniques:
  - ESP32: Light sleep
  - Monarch 2: Power Saving Mode (PSM) and extended Discontinuous Reception (eDRX)