Master's thesis

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

Bernt Johan Damslora

Data collection in a cellular sensor network with nRF9160

Master's thesis in Cybernetics and Robotics
Supervisor: Geir Mathisen
June 2019

NTNU
Kunnskap for en bedre verden

Bernt Johan Damslora

Data collection in a cellular sensor network with nRF9160

Master's thesis in Cybernetics and Robotics
Supervisor: Geir Mathisen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology
and Electrical Engineering
Department of Engineering Cybernetics

Master Thesis

Candidate:	**Bernt Johan Damslora**
Course:	**TTK4550 Engineering Cybernetics, Specialization Project**
Thesis title (Norwegian)	**Innsamling av data i et mobilnett-tilkoplet sensornettverk med nRF9160**
Thesis title (English):	**Data collection in a cellular sensor network with nRF9160**

Work desctiption: In this thesis we want to investigate the properties and readiness level of low-power cellular data collection.

Nordic Semiconductor's nRF9160 SiP (System in Package) supports different IoT (Internet of Things)-focused cellular protocols. We want to use this chip to explore the properties (energy consumption / efficiency, scalability, bandwidth, range of communication inside and outside buildings, communication robustness) of the different protocols, as well as their usability in the Norwegian cellular network.

The application areas for the data acquisition will be power control in buildings, smart grid and communication within other critical infrastructures. A particular case of interest is a smart grid experiment in Froan, Norway.

The tasks will be:

1. Conduct a literature search in the area of low-power cellular networks, focusing on the available implementations of these using the nRF9160 SiP.
2. Implement a data collection system based on the nRF9160 with the following properties:
    a. Nodes can transmit one or more sensor readings every 2.5 seconds
    b. Nodes time-stamp the reading with a common absolute time format with a precision of at least 1 second
    c. Data can be received by a different system in a common serialized format over a common protocol (e.g. JSON or a more efficient protocol over MQTT)
    d. Control messages will be transmitted to the nodes
3. Evaluate the usability and properties of the data collection system, with the following points of interest:
    a. Amount of data that can be transmitted reliably each 2.5 second interval
    b. Total power usage, and power efficiency
    c. Effect of increasing the number of nodes
    d. Coverage indoors and outdoors in selected locations

Start date:	January 1st, 2019
Due date:	June 3rd, 2019
Thesis performed at:	Department of Engineering Cybernetics
Supervisor:	Professor Geir Mathisen

Preface

This master thesis was carried out at the offices of Nordic Semiconductor. Specifically, Nordic provided hardware for development (nRF9160 development kit), as well as guidance in programming for and configuration of their hardware. The literature search, design, implementation and testing of the system was done by the author as part of this thesis. The supervisor, Geir Mathisen, provided valuable feedback and guidance during the thesis work.

This thesis is not a direct continuation of the author's project thesis. It does, however, use the software developed in the project thesis as a module in order to achieve one of the requirements.

Some parts of the application software used in the system is based on existing code from other projects. Specifically, the following source files are not written from scratch as part of this thesis work:

- *estimator.c* and its corresponing header *estimator.h* is imported as-is from [1], the author's specialization project.
- *main.c*, is based on the *mqtt_simple* example from the Nordic Semiconductor SDK, with major modifications.
- *sntp_raw.c* and its corresponing header *sntp_raw.h* is based on a library from Zephyr, with very minor modification.
- *sntp_pkt.h* defines a packet format and is imported as-is from the Zephyr project.

Otherwise, the application software has been written by the author as part of this thesis.

Abstract

New standards and products are being produced to satisfy an increasing demand for distributed data collection and control. In this thesis, we explore recent cellular standards from 3rd Generation Partnership Project (3GPP). Further, we design, implement and test a data collection node based on the Nordic Semiconductor nRF9160 System in Package (SiP). The data collection node is shown to fulfill several desirable requirements for an Internet of Things (IoT) use case, and further possible improvements and applications are discussed.

Sammendrag

Nye standarder og produkter blir for tiden utviklet for å dekke et økende behov for distribuert innsamling av data og styring. I denne oppgaven utforsker vi nye mobilnett-standarder fra 3GPP. Vi utvikler og tester et system for innsamling av data basert på nRF9160 SiP fra Nordic Semiconductor. Vi viser at systemet oppnår flere ønskelige krav i en IoT-sammenheng, og diskuterer mulige forbedringer og anvendelser.

Contents

Acronyms

3GPP 3rd Generation Partnership Project

ADC analog to digital converter

API Application Programming Interface

DK development kit

eDRX extended discontinuous reception

E-UTRA Evolved Universal Terrestrial Radio Access

I/O Input/Output

IoT Internet of Things

ISR interrupt service routine

LED light emitting diode

LTE Long-Term Evolution

LTE-M Long-Term Evolution Machine Type Communication

MCU microcontroller unit

MQTT Message Queuing Telemetry Transport

NB-IoT Narrowband IoT

PC personal computer

RTOS real-time operating system

SDK software development kit

SiP System in Package

SNTP Simple Network Time Protocol

UE User Equipment

USB Universal Serial Bus

List of Figures

Listings

1 Introduction

1.1 Background and motivation

Over the last decades, a trend has persisted of adding a high degree of computer monitoring and control to existing systems – or in marketing terminology, adding the word "smart". This has been possible and practical because year by year, better low-cost and low-power computer systems are developed and reach the market.

In any monitoring or control system, communication with sensors for data collection and actuators for control is essential for good algorithms to be useful. In the case of a geographically large system, like monitoring and control of a power grid (*smart grid*) or infrastructure in general, building a network for such communication can be costly and inconvenient. In some cases, the Internet is used in the communication, giving rise to the term *IoT*

Nordic Semiconductor has recently developed and released a low-cost and low-power single-package computer system with a cellular modem, to enable the use of the existing cellular infrastructure for such cases. That product, the nRF9160 SiP, connects using the recent cellular communication standards Narrowband IoT (NB-IoT) and Long-Term Evolution Machine Type Communication (LTE-M).

1.2 Scope and limitations

In this thesis, we explore the new cellular standards theoretically through 3GPP's specifications, and we explore their implementation in nRF9160 through its documentation. Further, we describe and document the development of a cellular platform that can be used as a basis for an Input/Output (I/O) node in a smart grid experiment or other IoT experimentation. The requirements are laid out based on desirable properties in a smart grid scenario: regular transmission of measurements with precise time stamps, two-way communication for control of loads or switches. A transmission period of 2.5 seconds is desired to match certain smart meter equipment.

We document the design, implementation and testing of the system. We discuss usability and properties of the system, including data capacity, power usage, power efficiency, scalability and cellular coverage.

The acquisition of the actual sensor data and actions to take when receiving commands are outside the scope. Examples are provided in the form of transmitting a calculated time stamp and remotely turning on and off an indicator light emitting diode (LED). These examples are in separate modules (source files), and the relevant behaviour can be implemented as needed for an experiment.

The software development resources available for the newly launched nRF9160 SiP – specifically Nordic Semiconductor's as well as the Zephyr project's libraries and tools – has rapidly and significantly evolved during the course of the project. The latest version was formally released as a stable version (nRF Connect SDK 0.4.0) in May 2019. To make the experience from this thesis more relevant as a basis for future projects, an effort was made to ensure the workflows described in this thesis, as well as the finished system itself, would work with this newest version. This reduced the time available for quantitative testing and optimization work. The functionality has only been verified in LTE-M mode (as opposed to NB-IoT), and power consumption is not optimized.

1.3 Main contribution

This thesis provides a basis for a low-power cellular edge-node in order to experiment with using distributed embedded systems in a smart grid or other use case. Furthermore, practical data rate, power consumption and coverage is evaluated, and can provide some insight into the state of the art and readiness level of cellular communication for this use case.

Parts of this thesis – specifically Chapter 5 – may also be useful as an example case for those wanting to implement other firmware using the Zephyr operating system and tools, especially on Nordic Semiconductor hardware.

1.4 Structure of this report

The rest of this report is structured as follows: The theoretical background to the thesis is presented in Chapter 2. The chapter starts with an exploration of the 3GPP cellular technologies that focus on IoT applications, entering into the details of the different specifications (LTE-M and NB-IoT). Further, the chapter covers the relevant details of the nRF9160 SiP. Finally, a time synchronization method and a messaging protocol are introduced in the context of this thesis.

The rest of the report covers the implemented data collection system. The requirements are stated in Chapter 3. Chapter 4 details the design decisions, underlying libraries and code used in the project, as well as the structure and behaviour of the new software written. Chapter 5 goes on to document the process of implementing the software in C and programming the hardware.

In Chapter 6, test results for the different requirements and metrics are recorded. These results are further discussed in Chapter 7. Finally, Chapter 8 concludes the thesis and proposes how future work can build on the results.

2 Background

2.1 3GPP cellular technologies

The 3GPP is a partnership of many telecommunications companies, with the goal of standardizing cellular radio communication (classically mobile phones). Each specification they release has many versions, grouped into feature stable *releases*. Each release describes new functionality and enhancements, and the releases are individually updates with corrections and clarifications. The specifications that will be relevant for this chapter are those describing the on-air radio access of Long-Term Evolution (LTE), Evolved Universal Terrestrial Radio Access (E-UTRA), as well as the requirements for equipment (called User Equipment (UE)) using that radio access. These can be found in the specifications numbered 36.xxx. [2]

LTE-M and NB-IoT are recent standards from the 3GPP for machine-to-machine communication in the licensed cellular bands. [3] LTE-M is an umbrella term for the efforts to make the existing E-UTRA useful for devices with lower complexity, bandwidth and power budgets. This manifests in the UE categories Cat-0, Cat-M1 and Cat-M2. These specifications were made to be compatible with the current E-UTRA radio technology, making the migration of infrastructure inexpensive, only needing software updates.

NB-IoT, on the other hand, was developed with the goal of specifying "a radio access for cellular internet of things, based to a great extent on a non-backward-compatible variant of E-UTRA, that addresses improved indoor coverage, support for massive number of low throughput devices, low delay sensitivity, ultra low device cost, low device power consumption and (optimised) network architecture." [4] Further, this new radio access was to be used in the LTE bands, as well as in bands formerly used for older cellular technology. In other words, NB-IoT was developed as a new specification, keeping in mind that it would coexist with E-UTRA, and leveraging the engineering, documentation and implementation efforts that had already gone into E-UTRA.

As the NB-IoT standard is very similar to normal E-UTRA and meant to be used alongside it, NB-IoT is described in the E-UTRA specifications. In the E-UTRA specification for modulation schemes and radio channels, there is a separate chapter at the end describing the modulation scheme of NB-IoT, heavily referencing the rest of the specification and borrowing much terminology. [5]

The main advantage of NB-IoT are derived from occupying less radio bandwidth for a single link than is possible with legacy LTE hardware (increasing the possible number of UEs per cell), as well as support for simpler modulation and less processing on the UE (enabling cheaper products). The increased indoor coverage of NB-IoT is specified in the form of a 8.3 dB higher maximum coupling loss target. [3] Simply put, the signal can be attenuated by that much more between the transmitter and receiver.

With the new IoT-focused specifications, 3GPP defines a new power saving feature to enable longer battery life: extended discontinuous reception (eDRX). This new mode enables the UE to disable radio reception for longer periods of time while remaining connected to the network, in the range of 5 seconds to multiple hours. [3]

2.2 Nordic Semiconductor nRF9160 SiP

The nRF9160 is a new SiP from Nordic Semiconductor, integrating an application processor, a cellular modem and several peripherals [6]. The cellular modem is compliant with 3GPP's specifications for UE categories M1 (release 13), NB1 (releases 13,14) and NB2(release 14). This means that it supports both the normal E-UTRA radio access, and the newer NB-IoT. The system also includes satellite navigation receiver hardware.

The package has built-in power management, and handles a supply voltage between 3.0 V and 5.5 V. 3.3 V is needed for compliance with the 3GPP specifications. I/O voltage is set separately, and can be between 1.7 V and 3.6 V. Importantly, this enables direct communication with 3.3 V equipment, but for 5 V equipment a logic level converter would be needed.

Nordic Semiconductor has released a hardware development board called PCA-10090 [7], pictured in Figure 1. The major active components on the board are:

- nRF9160 SiP, the central component
- nRF52840 System on Chip, routing some signals on the board and enabling Bluetooth and other 2.4 GHz connectivity (not used in this thesis)
- An interface microcontroller unit (MCU) used for communication with a computer

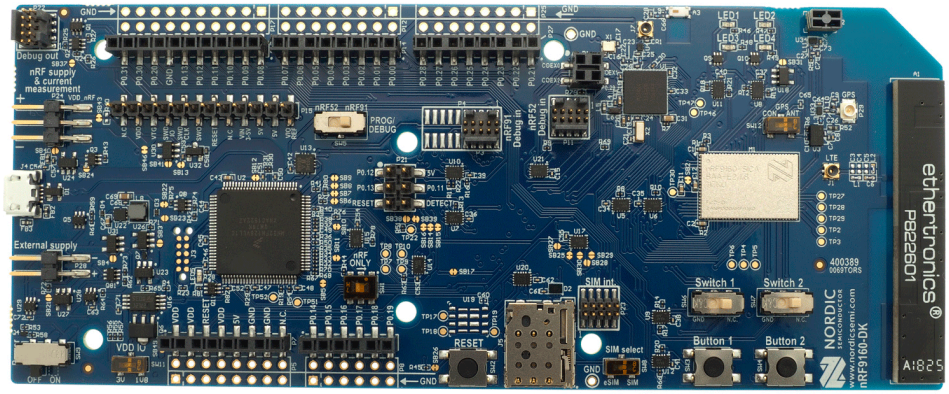


Figure 1: Photograph of the PCA-10090 board. The nRF9160 SiP can be seen as the silver colored package on the right side. Photo credit: Nordic Semiconductor

This board exposes the I/O pins on the SiP both as female headers and through-hole solderable connections. For the purposes of this thesis, these available connections make it easy to connect any sensor or actuator equipment. The dimensions and spacing of a subset of the headers match those of the well-known Arduino platform, enabling physical compatibility with Arduino add-on boards. The nRF9160 has a built in analog to digital converter (ADC), so appropriately scaled analog measurements can be taken without external hardware.

The development board enables measurement of the current passing to the nRF9160 power supply by way of exposed male headers. By cutting the SB43 solder bridge and connecting a current meter to the P24 set of male headers, the current can be measured. [8]

For software development, Nordic Semiconductor supplies the nRF Connect software development kit (SDK), which includes their fork of the Zephyr real-time operating system (RTOS). [9] Zephyr is a lightweight RTOS written to run on a number of microcontroller-based systems. [10] It provides kernel services like scheduling, mutexes, semaphores, and POSIX-like sockets for network communication. It also provides drivers with a common API for many peripherals, like the ADC or Serial Peripheral Interface. Crucially, Zephyr can be configured to include only the features needed for any given project, saving computation and memory resources.

2.3 Acquiring a common time reference

There are many approaches to achieving a synchronised time reference in distributed embedded systems. Using the time data from navigation satellites is one option, while synchronizing over the network is another. Navigation satellite time is very precise, but has the drawback of not being usable in areas where the signal is too attenuated (i.e. indoors). Tiny-sync is a network synchronization scheme that yields close to optimal synchronization with low requirements for memory and computation. [11] A C implementation of the Tiny-sync algorithm was implemented as part of [1], and is used in this thesis. The method used is an application of the Tiny-sync algorithm, with an Simple Network Time Protocol (SNTP) server as the common reference.

2.4 The MQTT protocol

Message Queuing Telemetry Transport (MQTT) is a protocol used for transferring messages across a network. The protocol is designed to be "light weight, open, simple, and designed so as to be easy to implement." [12] The protocol has become something of a de-facto standard in IoT use cases, and there are multiple software servers (often called *brokers*) and clients available. This allows for greater interoperability and easy integration with other experiments. In this thesis, the MQTT protocol is used to transfer messages to and from the data collection node. The Mosquitto project [13] contains both a client and server, and is used for testing in this thesis.

3 Requirements

3.1 Requirement 1: Hardware Platform

Part of the purpose of this thesis is to explore the properties of cellular protocols using the nRF9160 SiP. Thus, that is the chip that is used. For the purposes of this thesis, that translates into a requirement to use Nordic Semiconductor's development kit (DK) for said SiP, the board called PCA-10090 [7]. At the time of writing, this board is the only available development board with the nRF9160.

3.2 Requirement 2: Transmission Rate

The system shall transmit one or more readings every 2.5 seconds. This period is somewhat arbitrary, but lines up well with transmission rates of some electrical smart meters, and is fast enough that it can be used for more fine-grained control of production and consumption in a smart grid setting.

For example, a node could inform a central computer that more power is being used, or even will be used in a few seconds, and the central computer could react by sending a request or command to reduce consumption somewhere else.

3.3 Requirement 3: Synchronized accurate time stamps

The system shall transmit a time value along with the data in some common absolute time format, referring the time at which the measurement was taken. This value shall be accurate with a maximum error of 1 second, compared to the common reference.

While not directly useful for real-time monitoring of the latest state of a smart grid or other application, time-stamped data is essential for logging in experiments. The time coordinate also allows for temporal patterns to be noticed, enabling prediction of future values.

3.4 Requirement 4: Interoperability

The system shall transmit and receive data over a common format over a common protocol. MQTT is suggested. For a system used in an experimental setting, easy integration with custom solutions is an important point.

3.5 Requirement 5: Control messages

The system shall be able to receive, process and act on control messages from a central computer. This enables the node to be used not just as an observer, but also as an actuator. In the case of a smart grid, this could mean toggling or regulating the power to heavy loads.

The response time is not specified, but a quick response to this kind of input would naturally allow tighter control in a smart grid use case.

4 Design of the data collection node

4.1 Context and external interfaces

Since the data collection node is primarily intended for experimental use, the context in which it will be used may vary significantly. However, following are assumed to be the case in light of the requirements:

- There will be one or more sensors to measure some value(s), connected through some analog or digital pin(s)
- There will be some computer collecting the data for monitoring or further offline processing. This computer will also send control commands to the node
- There will be a time server used for time synchronization

The sensor and actuator connection are left unspecified. The protocol used for communication with the monitor and control computer is MQTT, as suggested in Chapter 3. The time synchronization is done through the SNTP protocol. Any protocol allowing the transfer of a time value could in principle be used, but SNTP has a significant advantage: there are many public time servers available that respond to SNTP requests, eliminating the need to run one's own accurate clock.

With the protocols decided, the context of the data collection node is as presented in Figure 2. Note that there can be multiple data collection nodes. In fact, that is the expected case in an experiment where a larger system such as an electric grid is monitored. In that case, the different data collection nodes publish messages to and read messages from unique MQTT topics.

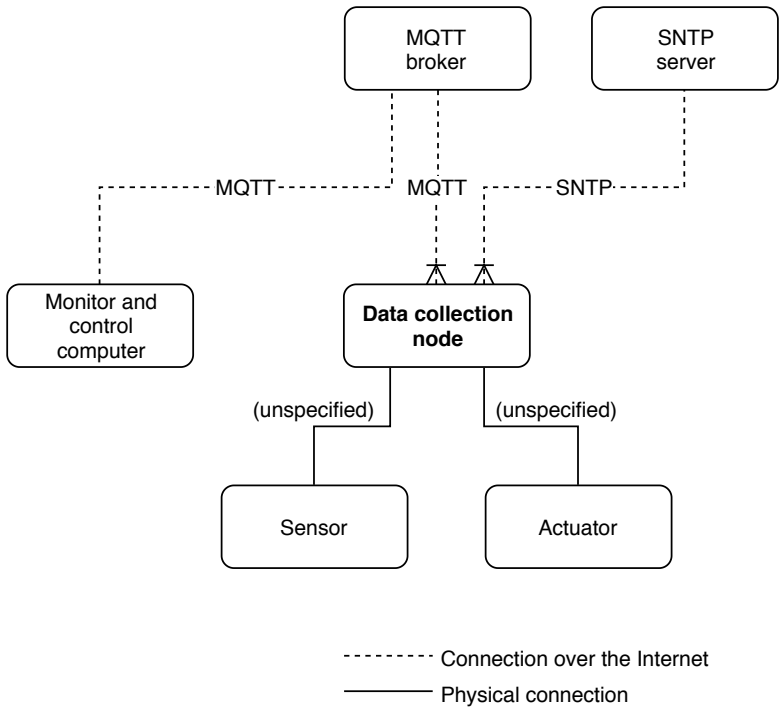


Figure 2: Context of the data collection node. Note that there can be multiple data collection nodes.

4.2 Hardware

As mentioned Chapter 3, the PCA-10090 board is the only available development board with the nRF9160. This makes it the obvious choice as the hardware platform for this thesis work. With the intended use being experimental, the board also has some noticeable advantages:

- The Universal Serial Bus (USB) connection and interface MCU enables debugging, monitoring or reprogramming in the field, requiring no equipment except a computer and a USB cable.
- The built-in indicator LEDs can be used for debugging or status indication
- The built-in switches and buttons can be used for configuration
- The exposed I/O pins in the form of female headers and through-hole solderable connections facilitate the temporary or permanent connection of additional hardware.
- There are exposed headers to measure the nRF9160s current draw.

There are also some disadvantages to the board, compared to a simpler layout containing only the nRF9160 package and the necessary components:

- The cost is higher than necessary, which may be a problem for large-scale uses.
- The components on the board may use additional power.
- The board is larger than necessary, which may be a problem in some scenarios.

These points are largely non-issues for experimental use. Regardless of their applicability, these negative points remain moot as long as there is no alternative. Should the need arise, a custom board could be designed.

4.3 Software

The software framework underlying the software written in this thesis is the nRF Connect SDK, including the Zephyr RTOS. Using the Zephyr RTOS and the Nordic Semiconductor-developed driver for the LTE modem is the most practical way to connect to the Internet. Furthermore, Zephyr includes an MQTT client library and SNTP client library, enabling the use of these protocols without writing low-level code.

Useful for this project specifically is existing boilerplate code to set up a simple lightweight MQTT client. This example uses Zephyr's MQTT client library and network stack to echo back any received MQTT message. The example is used as the basis for the event manager module developed in this thesis.

The software that needs to be implemented is divided into 4 modules: an event manager, a wall time module, a periodic task and a responsive task. The event manager (main.c) has the responsibility of initializing the system and waiting for events. The events that are expected, as well as their consequences are:

- A timer with a period of 2.5 seconds, triggering a sensor reading from the periodic task module (sensor.c).
- An incoming MQTT message, triggering the responsive task (controller.c)
- A timer with a period of the mqtt keep-alive interval, triggering MQTT maintenance operations (MQTT client library)

The wall time module (walltime.c), responsible for an accurate time estimate, will be called from the event manager under initialization. Subsequently, it will be called from the periodic task module whenever an accurate time stamp is needed. It needs functions for:

- synchronizing with the time master and initializing the Tiny-sync algorithm
- re-synchronizing with the time master and updating the Tiny-sync algorithm
- getting a time estimate

The periodic task module (sensor.c), responsible for the periodic measurements, will be called from the event manager, and make a measurement. This measurement part will vary based on the application, and is not clearly defined in this design. The module will then call the wall time module (walltime.c) to retrieve an accurate time stamp, pack the information together, and return an MQTT message to the event manager, ready to be sent out. The module only has this one function.

The responsive task module (controller.c), responsible for handling the control messages, will be called from the event manager with an MQTT message describing the work to do. This action will vary based on the application, and is not clearly defined in this design.

The software architecture for the data collection node is shown in Figure 3, which also shows the relation between the application software written for the data collection node and external dependencies.

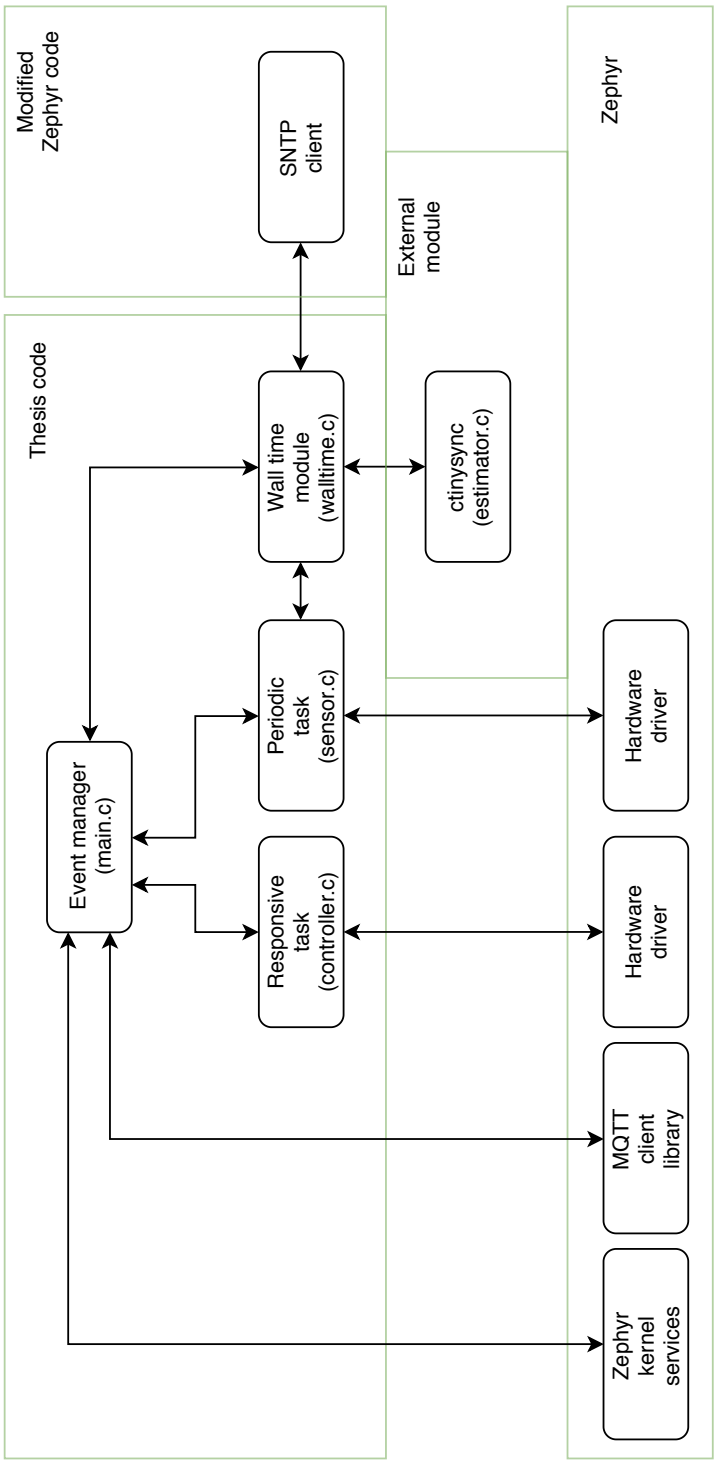


Figure 3: Software architecture of the data collection node

5 Implementation of the data collection node

5.1 The development environment

Setting up the environment for developing for the nRF9160 can be done on all major personal computer (PC) operating systems: Windows, Linux and MacOS. The process is documented in detail in [14]. During this thesis, most of the work was done on a Windows PC, so that is the platform that will be covered in this section. The software needed for the environment and their sources are listed in Table 1. The software needed is the same on all platforms (except for Chocolatey, the Windows package manager), but the way to install them varies.

A set of environment variables are needed. The file *%userprofile%\zephyrrc.cmd* must be created with the following content (if the GNU Arm Embedded Toolchain was installed somewhere else, that path would have to be used instead):

```
set ZEPHYR_TOOLCHAIN_VARIANT=gnuarmemb
set GNUARMEMB_TOOLCHAIN_PATH=c:\gnuarmemb
```

With the development environment ready, the nRF Connect SDK itself can be downloaded. After creating a working directory and entering it, this is done with the west tool with the following commands:

```
west init -m https://github.com/NordicPlayground/fw-nrfconnect-nrf --mr v0.4.0
west update
```

Once the SDK is downloaded, more specific dependencies need to be installed. This is done with the following commands:

```
pip3 install -r zephyr/scripts/requirements.txt
pip3 install -r nrf/scripts/requirements.txt
pip3 install -r mcuboot/scripts/requirements.txt
```

Each time the SDK is used in a console, the file *(SDK root)/zephyr/zephyr-env.cmd* must be run to set up the correct environment variables. At this point, the SDK is ready to be used.

Software	**Needed for**	**Source / Install command on Windows / Comment**
Chocolatey	Package installation	[15]
Cmake	Compilation of Zephyr	`choco install cmake -installargs 'ADD_CMAKE_TO_PATH=System'`
Git	Zephyr version control	`choco install git`
Python	Compilation of Zephyr	`choco install python`
Ninja	Compilation of Zephyr	`choco install ninja`
Device Tree Compiler	Compilation of Zephyr	`choco install dtc-msys2`
Gperf	Compilation of Zephyr	`choco install gperf`
GNU Arm Embedded Toolchain	Compilation for nRF9160	[16] Select version 7-2018-q2-update. Install to C:\gnuarmemb.
West	Zephyr version control	pip3 install west
Segger J-Link Software	Programming nRF9160	[17]
nRF5x Command-Line Tools	Programming nRF9160	[18]

Table 1: Software needed to compile for and program the nRF9160 with the nRF Connect SDK.

The attached thesis source code folder *gridnode* can be placed inside the folder *(SDK root)/nrf/applications*. After navigating to the *gridnode* folder, the following commands are run to compile and program a connected nRF9160 DK:

```
mkdir build
cd build
cmake -GNinja -DBOARD=nrf9160_pca10090ns ..
ninja flash
```

After this has been done once, it is sufficient to run the command `ninja flash`. This building and programming process is the same for any example or application written for the nRF9160.

5.2 Event manager

As laid out in Section 4.3, the event manager is responsible for initialization and responding to the possible events, and is largely based on the MQTT simple example from the nRF Connect SDK. The module serves as the entry point for two threads: the main thread, and the MQTT poller thread. The initialization code is listed in Listing 5.1. The `printk` and `leds_write` statements are used for debugging purposes. The `if`-clauses check for errors, and return from the main function, rebooting the processor, if an error is encountered.

The function calls during this procedure can be broken down as follows:

- `leds_init`: Call to led.c to initialize indicator LEDs.
- `modem_configure`: Call to function imported from the MQTT example, to set up the modem and wait for an internet connection.
- `client_init`: Call to function imported from the MQTT example, to set up the MQTT software client.
- `mqtt_connect`: Call to function imported from the MQTT example, to connect to the MQTT broker.
- `fds_init`: Call to function imported from the MQTT example, to set up a file descriptor for the MQTT network socket.
- `k_poll_signal_init`: Calls to the Zephyr kernel, to set up two incoming signals. These signals are simple binary semaphores used to enable waiting for either an MQTT-related event, or the periodic timer. The signals are then arranged in the `events` structure for later reference.
- `walltime_init`: Call to the wall time module, to synchronize with the SNTP server and initialize the Tiny-sync algorithm.
- `k_timer_start`: Call to the Zephyr kernel, starting a timer with a period of 2.5 seconds.
- `k_poll_signal_raise`: Call to the Zephyr kernel, notifying the MQTT poller thread that it can start polling.

```
void main(void)
{
  int err;

  printk("The MQTT simple sample started\n");
  leds_init();

  leds_write(0xF,0xF);
  modem_configure();
  leds_write(0xF,0x0);

  client_init(&client);

  err = mqtt_connect(&client);
  if (err != 0) {
    printk("ERROR: mqtt_connect %d\n", err);
    return;
  }

  err = fds_init(&client);
  if (err != 0) {
    printk("ERROR: fds_init %d\n", err);
    return;
  }


  // Signals waking up the event handler
  k_poll_signal_init(&mqtt_event_signal);
  k_poll_signal_init(&periodic_event_signal);
  struct k_poll_event events[2] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &mqtt_event_signal),
    K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &periodic_event_signal),
    };

  walltime_init();

  k_timer_start(&periodic_id, K_MSEC(2500), K_MSEC(2500));
  k_poll_signal_raise(&mqtt_ready_signal, 0);
```

Listing 5.1: Excerpt from main.c showing the initialization procedure at the start of the main() function

```
    while (1) {
    leds_write(0x1,0x1);
    k_poll(events, 2, K_FOREVER);
    leds_write(0x1,0x0);

    // Handle periodic sending
    if (events[1].signal->signaled) {
      events[1].signal->signaled = 0;
      events[1].state = K_POLL_STATE_NOT_READY;

      err = sensor_build_payload(payload_buf);
      if (err < 0) {
        break;
      }

      data_publish(&client, MQTT_QOS_1_AT_LEAST_ONCE,
        payload_buf, strlen(payload_buf));
    }

    //Do MQTT operations
    if (events[0].signal->signaled) {
      events[0].signal->signaled = 0;
      events[0].state = K_POLL_STATE_NOT_READY;
      err = events[0].signal->result;
      if (err < 0) {
        printk("ERROR: poll %d\n", err);
        break;
      }

      // Connection maintainance
      err = mqtt_live(&client);
      if (err != 0) {
        printk("ERROR: mqtt_live %d\n", err);
        break;
      }

      // Handle MQTT socket events
      if ((fds.revents & POLLIN) == POLLIN) {
        err = mqtt_input(&client);
        if (err != 0) {
          printk("ERROR: mqtt_input %d\n", err);
          break;
        }
      }
      if ((fds.revents & POLLERR) == POLLERR) {
        printk("POLLERR\n");
        break;
      }
      if ((fds.revents & POLLNVAL) == POLLNVAL) {
        printk("POLLNVAL\n");
        break;
      }
    }

  }
```

Listing 5.2: Excerpt from main.c showing the infinite loop in the main() function

```
1 /* Timer for periodic send actions */
2 void periodic_isr (struct k_timer *timer_id); // Fwd declaration needed
3 K_TIMER_DEFINE(periodic_id, periodic_isr, NULL);
4 ...
5 void periodic_isr (struct k_timer *timer_id){
6   k_poll_signal_raise(&periodic_event_signal, 0);
7 }
```

Listing 5.3: Excerpt from main.c showing the lines necessary to raise a signal when the timer is triggered. In the source file, lines 1-3 are located near the top, while lines 5-7 are located near the bottom.

Once the initialization is complete, the main loop of the program starts. It is listed in Listing 5.2. Once again, `leds_write` is used for debugging. This loop starts by waiting for the return of `k_poll`. This function yields processor time to other threads until any signal it waits for is raised. The signals it waits for are those set up during initialization: one for MQTT related events, and one for the periodic timer. Depending on which signal wakes the thread, either lines 8-17 or lines 22-51 will be run. In each case, the signal is reset and the event is handled. The code within these handlers are further discussed in the following sections.

5.3 Periodic updates

To enable the periodic signal, a timer is started with a period of 2.5 seconds, as shown in Listing 5.1, line 41. This timer is defined and connected to an interrupt service routine (ISR) near the top of the source file using a macro from Zephyr, as shown in Listing 5.3. The ISR itself only contains a single function call to raise the signal.

Once the signal wakes the main thread, lines 8-17 in Listing 5.2 will run. First, the periodic task module is called with `sensor_build_payload`. The message is then sent to the MQTT driver to be published with the `data_publish` function, which is imported from the MQTT example.

The periodic task module is listed in its entirety in Listing 5.4. At the very start of its single function, the code for taking a measurement would be added in an actual use case. An accurate time stamp from the wall time module is requested with the function `walltime_get`. The uncertainty of the timestamp as reported by the wall time module is checked, and a re-calibration requested with the function `walltime_calibrate` if the uncertainty is too high. Finally, the time value is loaded into an MQTT message string with the standard `sprintf` function. In use, the sensor measurement would also be added to the string.

```c
#include "sensor.h"
#include "walltime.h"

int sensor_build_payload(u8_t * payload_buf){
    uint64_t timestamp = 0;
    uint64_t accuracy = 0;
    int err;

    // Get wall-time stamp
    err = walltime_get(&timestamp, &accuracy);
    if (err < 0) return -1;

    printk("accuracy(1024ths):%llu\n",accuracy>>22);

    // Recalibrate if inaccurate (>=0.5s)
    if (accuracy>>31 > 0){
        printk("TOO INACCURATE\n");
        err = walltime_calibrate();
        if (err < 0) return -1;
    }

    // Build payload string
    sprintf(payload_buf, "%x %x", (u32_t)(timestamp>>32), (u32_t)(timestamp));

    return 0;
}
```

Listing 5.4: Periodic task module (sensor.c) source code.

5.4 Control message handling

The MQTT poller thread is listed in Listing 5.5 To detect incoming MQTT messages, the socket set up for the MQTT driver can be checked for input. This can be done periodically, but to minimize response time, it is done as the idle activity in the program. This is why a second thread is used.

The MQTT poller thread starts by waiting for the ready signal sent at the end of the event managers initialization. This ensures that the MQTT socket is properly set up before it is polled. An infinite loop is then entered (Listing 5.5, lines 12-20). In this loop, waiting for incoming data on the MQTT socket is done through the Zephyr networks stack's `poll` function. The function will time out after the configured MQTT keep-alive interval. Once that function returns, either due to new data or due to a timeout, the signal is raised, waking the main thread.

To start a thread running the `mqtt_poller_thread` function at startup, the `K_THREAD_DEFINE` macro is used, as shown in Listing 5.5, lines 24-27. As opposed to the kernel's `k_poll` function, the network stack's `poll` does not yield processor time while blocking. Therefore, the MQTT poller thread is given a lower priority (i. e. a higher priority number) than the main thread. This allows the main thread to preempt the poll function and do its tasks.

Once the signal wakes the main thread, lines 22-51 in Listing 5.2 will run. The function `mqtt_live` from the MQTT driver is run whether there is new data or not, because it is responsible for sending keep-alive messages to the MQTT broker. If the socket has the flag POLLIN set, which means there is new data, `mqtt_input` from the MQTT driver is run, which will process the new data. This in turn will cause the configured callback `mqtt_evt_handler` in main.c to run. This callback is imported from the MQTT example and mostly left unchanged, with one significant modification: the responsive task module's `handle_message` function is called with the message. This allows the responsive task module to do whatever action is appropriate.

The periodic task module is listed in its entirety in Listing 5.6. This is where usage specific behaviour would be implemented. As an example, the module implements a simple decision tree based on each character in the MQTT message: If the first character is 'L', turn the indicator LED4 on/off based on whether the next character is '0' or '1'. This enables turning the indicator on or off by sending the messages "L0" or "L1", respectively. In a smart grid scenario, this could represent a load or a generator being controlled to balance production and consumption.

The presence of the MQTT poller thread does not entail that the rest of the modules have to be thread-safe, because it does not interact with any of the other modules. Further, because the MQTT poller thread has a lower priority, the main thread will never be preempted by it.

```
void mqtt_poller_thread(void){
  k_poll_signal_init(&mqtt_ready_signal);
  int err;
  struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &mqtt_ready_signal),
    };
  printk("mqtt_poller_thread waiting for go\n");
  k_poll(events, 1, K_FOREVER);
  printk("mqtt_poller_thread got go\n");
  while(1){
    leds_write(0x2,0x2);

    err = poll(&fds, 1, K_SECONDS(CONFIG_MQTT_KEEPALIVE));

    leds_write(0x2,0x0);
    printk("poll revents 0x%x\n", fds.revents);
    k_poll_signal_raise(&mqtt_event_signal, err);
  }

}

#define STACKSIZE 1024
#define PRIORITY 8 // Less urgent than main thread (7); BSD poll does not yield
K_THREAD_DEFINE(mqtt_poller_thread_id, STACKSIZE, mqtt_poller_thread, NULL, NULL,
     NULL,
    PRIORITY, 0, K_NO_WAIT);
```

Listing 5.5: Excerpt from main.c showing the MQTT poller thread entry point and definition.

```
#include "controller.h"
#include "led.h"
void handle_message (u8_t * buf, size_t length){
    if (length < 2) return;
    switch (buf[0]) {
        case 'L':
            switch (buf[1]) {
                case '1':
                    leds_write(0x8,0xF);
                    break;
                case '0':
                    leds_write(0x8,0x0);
                    break;
                default:
                    printk("Unknown char [1]: %c\n",buf[1]);
                    break;
            }
        break;
    default:
    printk("Unknown char [0]: %c\n",buf[0]);
    break;
    }
};
```

Listing 5.6: Responsive task module (controller.c) source code.

```
int walltime_init(){
    tinysync_est_state_t_initialize(&state);
    last_result = TINYSYNC_EST_INVALID;
    int err = walltime_calibrate();
    if (err < 0) return -1;
    return 0;
}
```

Listing 5.7: Excerpt from walltime.c showing the initialization procedure.

5.5 Time estimation

The role of the wall time module is to get a time reference from the network with SNTP, and to use the implementation of the Tiny-sync algorithm imported from [1] to create accurate time estimates based on that reference. The Tiny-sync implementation (estimator.c) outputs constraints on the linear relation between two timers, so these constraints also have to be used to calculate the estimates.

The initialization procedure for the wall time module is listed in Listing 5.7. It simply initializes the Tiny-sync algorithm and goes on to perform a first calibration. An error is returned if calibration is not possible. The static variable `last_result` is used to keep track of the state of the estimate (i.e. if the estimate is valid).

```
int walltime_calibrate(){
    tinysync_datapoint_t datapoint;
    int err = get_datapoint(&datapoint);
    if (err < 0) return -1;
    last_result = tinysync_est_etimate(&state, &datapoint);
    while(last_result != TINYSYNC_EST_OK){
        k_sleep(K_MSEC(1000));
        int err = get_datapoint(&datapoint);
        if (err < 0) return -1;
        last_result = tinysync_est_etimate(&state, &datapoint);
    }
    return 0;
}
```

Listing 5.8: Excerpt from walltime.c showing the calibration procedure.

```
    datapoint->t_o = k_uptime_get();
    rv = sntp_request(&ctx, K_FOREVER, &(datapoint->t_b) );
    if (rv < 0) {
        printk("Failed to send sntp request: %d", rv);
        return -1;
    }
    datapoint->t_r = k_uptime_get();
```

Listing 5.9: Excerpt from walltime.c showing the generation of a Tiny-sync datapoint.

The calibration procedure for the wall time module is listed in Listing 5.8. The `get_datapoint` function is used to retrieve the set of 3 sequential time values needed by the Tiny-sync algorithm. a while-loop is employed to make sure that the algorithm is run with new data points until a valid set of constraints are generated. Normally, only 2 data points are needed, but in case of sudden non-linearity between the clocks, more may be needed. The essential part of the `get_datapoint` function is listed in Listing 5.9. The first and last time values are generated by the local system clock, accessed by `k_uptime_get`, whereas the middle time value is retrieved from the SNTP server.

When another module retrieves an accurate time estimate from the wall time module, this is done with the `walltime_get` function, listed in Listing 5.10. The maximum and minimum value of the reference clock is calculated. The mean of the minimum and maximum values is the estimated time, and half the difference is the maximum error. These values are returned to the caller through pointers provided as arguments.

```
int walltime_get(uint64_t * t_expected, uint64_t * accuracy){
    if (last_result != TINYSYNC_EST_OK) {
        return -1;
    }
    uint64_t now = k_uptime_get();
    uint64_t max_t_2 = (uint64_t)((((double)now) - state.lineset.ba.b) / state.
     lineset.ba.a);
    uint64_t min_t_2 = (uint64_t)((((double)now) - state.lineset.ab.b) / state.
     lineset.ab.a);
    *t_expected = (max_t_2/2) + (min_t_2/2);
    *accuracy = (max_t_2/2) - (min_t_2/2);
    //printk("T2EXP:%llx\n", (max_t_2/2) + (min_t_2/2) );
    //printk("ACC (1024ths)(1sec):%lld\n", ((max_t_2/2) - (min_t_2/2)) >> 22 );
    //printk("RESLT:%d\n",result);
    return 0;
}
```

Listing 5.10: Excerpt from walltime.c showing the estimation of the current time.

6 Testing and results

6.1 Configuration and test environment

As can be seen in figure Figure 2, three Internet-connected computers are necessary to test the system in its context: An SNTP server, an MQTT broker and a computer for monitoring and control, running an MQTT client. These computers could in principle also be one and the same. In order to test the system, the Mosquitto MQTT broker was set up on a personal server, and the Mosquitto MQTT client was installed on a PC for monitoring and control. The host name of the MQTT broker to use is set in the *prj.conf* configuration file. The University of Oslo's time server, *ntp.uio.no*, was used for the SNTP requests. An attempt was made to communicate over the NB-IoT standard by setting `CONFIG_LTE_NETWORK_MODE_NBIOT=y` in the *prj.conf* configuration file.

Power measurement was done as described in [8], connecting a calculating multi-meter to the development board, and measuring the average current draw until it stabilized. Measurement of signal strength was done by using the LTE Link Monitor application provided by Nordic Semiconductor, documented in [19].

6.2 Functionality

Each requirement was tested separately. The details and results are presented in this section. Requirement 1 did not need testing, as it is not a functional requirement.

6.2.1 Requirement 2: Transmission Rate

The transmission rate was tested by counting the transmissions (visible on debug output) during the course of 1 minute. 24 transmissions were counted, confirming approximately 24 transmissions per minute, or one each 2.5 seconds.

6.2.2 Requirement 3: Synchronized accurate time stamps

The accuracy of the timestamps were not measured directly, but the Tiny-sync algorithms maximum and minimum values were used to calculate the maximum error. This value over time is shown in Figure 4, and is always within 1 second.

6.2.3 Requirement 4: Interoperability

Interoperability with other MQTT clients was tested by using the Mosquitto MQTT client to read messages from the node, and to send control messages to the node. The messages were both sent and received successfully.

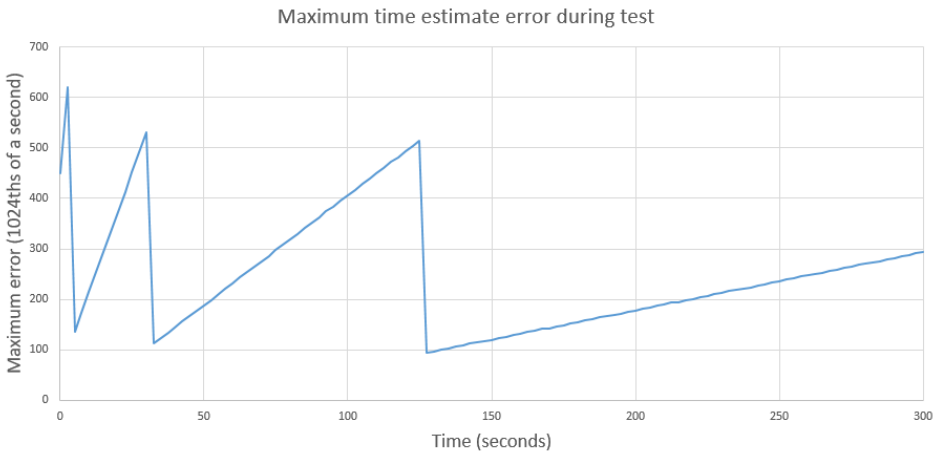


Figure 4: Development over time of the maximum error of the time estimate. The sudden reductions represent synchronization events.

Payload size (bytes)	Current consumption (mA)	Transmission success
0	13.96	Yes
4	15.99	Yes
16	16.76	Yes
64	16.55	Yes
512	16.90	Yes
4096	18.62	Yes

Table 2: Test results from increasing payload size.

6.2.4 Requirement 5: Control messages

In order to test the control message functionality, the strings "L0" and "L1" was sent to the node to control the state of the assigned indicator LED. The LED responded as expected.

6.3 Quantitative properties

6.3.1 Power usage

The payload size was increased in various steps from 0 bytes to 4096 bytes, and the current consumption of the nRF9160 was measured. The data was transmitted successfully and regularly in all cases. The results are presented in Table 2. The voltage across the power supply was measured as 4.92 V in every case

Location	Signal strength, LTE-M (dBm)	Signal strength, NB-IoT (dBm)
Third floor	-86	-80
Ground floor	-93	-85
Sub-basement	-114	-105

Table 3: Variation in signal strength by location and cellular mode

6.3.2 Coverage

By using the LTE Link Monitor application provided by Nordic Semiconductor, the signal strength was measured in a concrete building, on three different floors. The third floor, ground floor and sub-basement. The results are presented in Table 3. In LTE-M mode, the data was successfully transmitted in each of the tests. In NB-IoT mode, the node failed to connect to the network in all of the tests. When the signal strength fluctuated, the midpoint of the maximum and minimum values was used.

7 Discussion

Based on the results, the system fulfills each requirement. The transmission rate was correct, the time stamps were accurate, the MQTT protocol was employed, and control messages were handled correctly. Further, as shown in Table 2, message payload sizes of up to 4 kilobytes were tested successfully. This is equivalent to 1024 32-bit numbers, if encoded efficiently. Even if encoded as machine precision strings of number characters, there is capacity for hundreds of data points in a single transmission.

The coverage test yielded promising results, showing full functionality in a sub-basement. The higher numbers for the NB-IoT mode suggest that coverage would indeed be better in this mode, but as there was a network connection error while testing NB-IoT, this cannot be confirmed. The higher signal strength suggest that the issue was with configuration or network registration, and not a coverage issue. In a use case where the node is used to control loads in technical indoor rooms, like water heaters, this point is especially relevant.

The coverage test did not address the issue of having multiple nodes within the same location. As the massive increase in clients per location is part of what the NB-IoT and LTE-M standards address, this should not be a problem, especially in NB-IoT mode with its lower occupied bandwidth. However this should be tested to obtain practical results. In an application like a solar farm with a node on each panel, the density should be considered and tested.

It is desirable to minimize power consumption in any application, but it is especially relevant in battery powered applications, like a remote cottage or a stand-alone sensor. The power consumption, presented in Table 2 does not scale linearly with the message size. There are several reasons why the power consumption could be much higher than what is theoretically possible:

- Power saving features of LTE-M and NB-IoT are not configured.
- Debug messages are being sent out of the nRF9160, in addition to the radio activity.
- The nRF9160 is constantly listening for control messages.

A significant improvement in power consumption may therefore be achieved by doing the following:

- Configuring the modem to use power saving features, like eDRX.
- Disabling debug communication

- Checking for control messages at set intervals.

The MQTT protocol provides great opportunities for interoperability and provides useful guarantees, but its overhead, while low, may be significant for low-power applications. For experimental use, the interoperability and ease of implementation is probably worth the extra cost, but for researching the limits of power-efficient communication, using a lower layer protocol directly may be advantageous.

The data collection node was not tested with any actual sensor data or actuators. In actual usage, such hardware would be attached. Testing the system with external hardware, though it should not influence the results directly, may be useful to discover any issues that may arise, such as interference or processing delays.

Last, but not least, the issue of security needs to be addressed. To facilitate faster development and simpler configuration, the network communication used in this thesis is unencrypted and unauthenticated. This means the system, as implemented, would be vulnerable to, amongst others, a bad actor sending false data or commands to the MQTT broker. For data collection purposes in research, this would mainly be an annoyance, but if the system is connected to actuators that may impact safety, it is unacceptable.

Because the system is Internet-connected by nature, using a closed private network to mitigate the security issues is not an option. In order to secure the system, Transport Layer Security could be used. There is support for this in Zephyr and the nRF Connect SDK. The security features of MQTT could also be explored.

8 Conclusion

The data collection node was implemented to the satisfaction of the requirements laid out in Chapter 3. We can conclude that the nRF9160 can be used as a basis for a cellular data collection system, able to report relatively large amounts of data (in an IoT context) from outdoor as well as deep indoor environments. Network-based time proved successful as a way to synchronize time stamps from a node with a known common time reference.

8.1 Future work

There are several future avenues to explore based on the results, as discussed in Chapter 7:

- The limits of data transmission with the nRF9160 can be tested further.
- The advantages or disadvantages of NB-IoT over LTE-M could be evaluated in practice by setting up a test environment were other variables are kept similar.
- The limits of the radio coverage can be tested further, especially the limits of saturating a location with clients.
- Power consumption can be further optimized through various means.
- The security of the system can be improved.
- External sensors or actuators can be implemented and tested.

Sources

[1] Damslora, B. J. 2018. Bluetooth Mesh and low-power data acquisition in real time. Specialization project report, NTNU subject TTK4550, fall 2018.

[2] 3GPP. 2019. 3gpp specification series: 36series. https://www.3gpp.org/DynaReport/36-series.htm. Accessed: 2019-05-31.

[3] 3GPP. 2016. Standards for the IoT. http://www.3gpp.org/images/presentations/2016_11_3gpp_Standards_for_IoT.pdf. Accessed: 2018-12-02.

[4] 3GPP. 2015. RP-151621: New WI proposal: NB-IoT. https://portal.3gpp.org/ngppapp/CreateTDoc.aspx?mode=view&contributionUid=RP-151621. Accessed: 2019-05-31.

[5] 3GPP. 2019. 36.211: Evolved Universal Terrestrial Radio Access (E-UTRA); physical channels and modulation. http://www.3gpp.org/ftp//Specs/archive/36_series/36.211/36211-ea0.zip. Version 14.10.0. Accessed: 2019-05-31.

[6] Nordic Semiconductor. 2019. nRF9160 documentation. https://infocenter.nordicsemi.com/topic/struct_nrf91/struct/nrf9160.html. Accessed: 2019-04-15.

[7] Nordic Semiconductor. 2019. nRF9160 Development Kit documentation. https://infocenter.nordicsemi.com/topic/ug_nrf91_dk/UG/nrf91_DK/intro.html. Accessed: 2019-04-15.

[8] Nordic Semiconductor. 2019. nRF9160 Development Kit documentation: Measuring current. https://infocenter.nordicsemi.com/topic/ug_nrf91_dk/UG/nrf91_DK/hw_measure_current.html. Accessed: 2019-04-15.

[9] Nordic Semiconductor. 2019. nRF Connect SDK documentation, version 0.4.0. https://developer.nordicsemi.com/nRF_Connect_SDK/doc/0.4.0/nrf/index.html. Accessed: 2019-05-24.

[10] Zephyr project members and contributors. 2019. Zephyr documentation - introduction. https://docs.zephyrproject.org/latest/introduction/index.html. Accessed: 2019-05-29.

[11] Yoon, S., Veerarittiphan, C., & Sichitiu, M. L. June 2007. Tiny-sync: Tight time synchronization for wireless sensor networks. *ACM Trans. Sen. Netw.*, 3(2). URL: http://doi.acm.org/10.1145/1240226.1240228, doi:10.1145/1240226.1240228.

[12] OASIS Standard. 2014. MQTT version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Accessed: 2019-05-31.

[13] Eclipse Foundation. 2019. Eclipse Mosquitto. https://mosquitto.org/. Accessed: 2019-05-31.

[14] Nordic Semiconductor. 2019. nRF Connect SDK documentation, version 0.4.0 - installing the nRF Connect SDK. https://developer.nordicsemi.com/nRF_Connect_SDK/doc/0.4.0/nrf/gs_installing.html. Accessed: 2019-05-24.

[15] Chocolatey Software, Inc. 2019. Chocolatey - installation. https://chocolatey.org/install. Accessed: 2019-05-31.

[16] Arm Limited. 2019. GNU Toolchain | GNU-RM Downloads. https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads. Accessed: 2019-05-31.

[17] SEGGER Microcontroller GmbH. 2019. Downloads - J-Link / J-Trace. https://www.segger.com/downloads/jlink/. Accessed: 2019-05-31.

[18] Nordic Semiconductor. 2019. nRF5 Command Line Tools. https://www.nordicsemi.com/Software-and-Tools/Development-Tools/nRF5-Command-Line-Tools. Accessed: 2019-05-31.

[19] Nordic Semiconductor. 2019. nRF Connect LTE Link Monitor documentation. https://infocenter.nordicsemi.com/topic/ug_link_monitor/UG/link_monitor/lm_intro.html. Accessed: 2019-05-31.

A API of the implemented and imported software modules

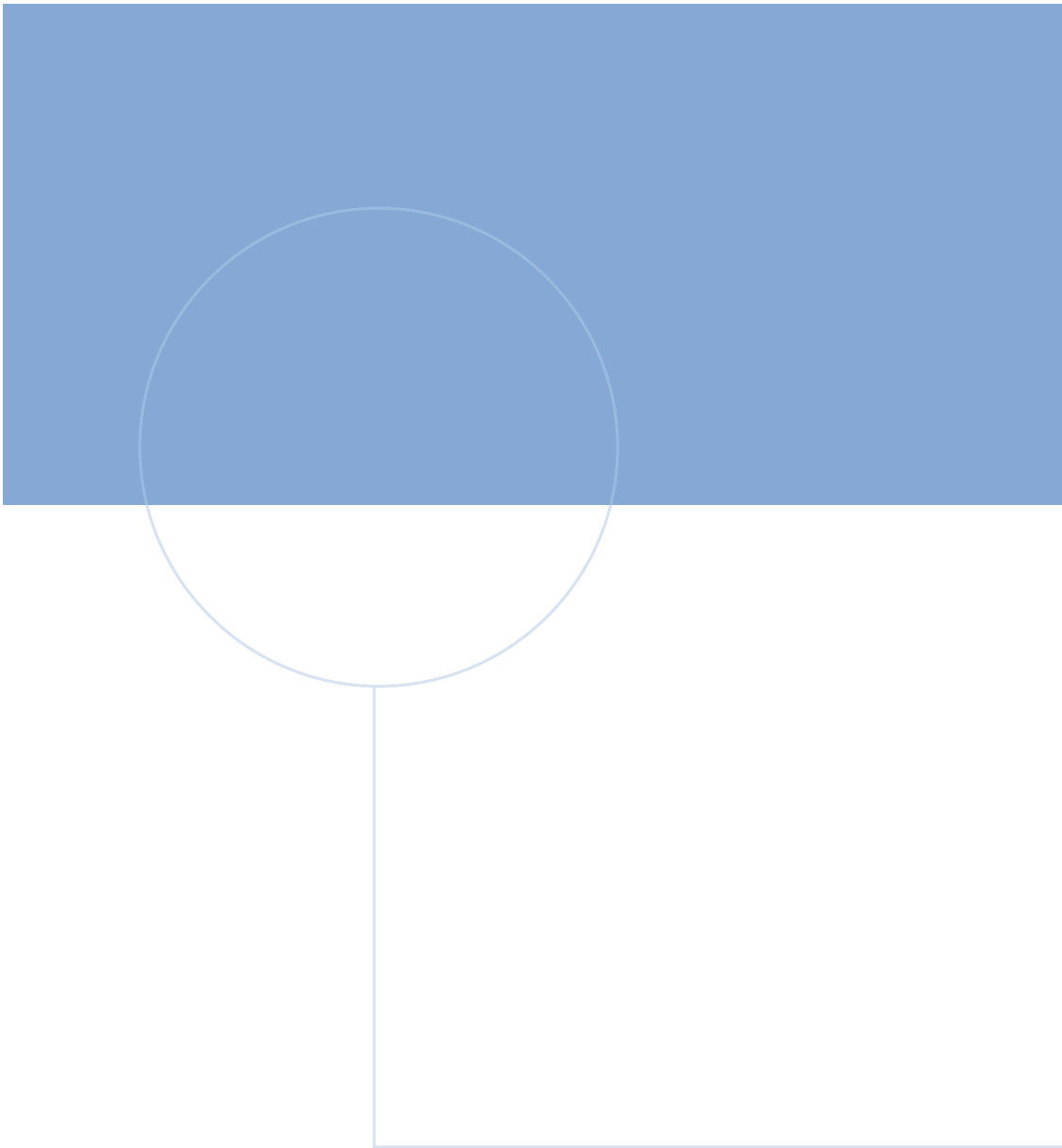
Table 4 lists all externally accessible functions in each software module written for and imported to this thesis.

Module	Function	Input(s)	Return	Result
controller.c	`handle_message`	string buffer	none	Appropriate actuation according to message
estimator.c	`tinysync_est_state_t_initialize`	Tiny-sync state	none	The Tiny-sync state is initialized
estimator.c	`tinysync_est_etimate`	Tiny-sync state, Tiny-sync datapoint	none	The Tiny-sync algorithm is run, updating the state
led.c	`leds_init`	none	none	The indicator LEDs are configured as outputs
led.c	`leds_write`	bitmask to write to, bitmask to enable	none	The indicator LEDs are updated according to input
sensor.c	`sensor_build_payload`	string buffer	none	The sensor message is loaded into the buffer
sntp_raw.c	`sntp_init`	sntp state, IP address	error code	The SNTP client is initialized
sntp_raw.c	`sntp_request`	sntp state, timeout value, time stamp pointer	error code	An SNTP request is sent, and the time from the server is stored in the supplied pointer
sntp_raw.c	`sntp_close`	sntp state	none	The SNTP client is closed
walltime.c	`walltime_init`	none	error code	The wall time module is initialized, and a first time calibration is performed
walltime.c	`walltime_calibrate`	none	error code	A time calibration is performed
walltime.c	`walltime_get`	time stamp pointer, accuracy pointer	error code	An estimate of the current time and its maximum error is written to the supplied pointers

Table 4: API of each software module

B Availability of the authors previous work

This thesis uses software from the author's specialization project. The project report and attachments are available from the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The author grants the department license to distribute the specialization project report [1] and the associated source code files without limitation.



NTNU

Kunnskap for en bedre verden