LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT

School of Energy Systems

Degree Programme in Electrical Engineering

Harri Rahikainen

DESIGN AND PROTOTYPE OF WEARABLE DATA ACQUISITION SYSTEM

Examiners: Professor Pertti Silventoinen

M.Sc. (Tech.) Antti Immonen

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Energy Systems
Degree Programme in Electrical Engineering

Harri Rahikainen

Design and Prototype of Wearable Data Acquisition System

Master's thesis

2021

55 pages, 12 figures, 6 table(s) and 0 appendices

Examiners: Professor Pertti Silventoinen
M.Sc. (Tech.) Antti Immonen

Keywords: Wearable, IoT, Sensor, Bluetooth Low Energy

The number of connected devices is increasing rapidly. The applications of the Internet of Things span many sectors of our society from intelligent infrastructure to industry and consumer electronics. Smart sensing devices are being used to measure the environment and even monitor human health and activity. The data that these smart devices produce needs to be transferred from the device to the Internet to be processed and viewed.

In this thesis, the design of a wearable data acquisition device that independently measures and stores multiple quantities and transmits the measured data is studied. First, the technical requirements are gathered and discussed, and then a prototype is implemented.

The device prototype was implemented on an embedded controller using a real-time operating system. The resulting prototype measures heart rate, temperature and relative humidity, and stores this data persistently. The prototype utilizes Bluetooth Low Energy to reliably send the stored data and status messages to a gateway device.

The analysis of the prototype found that the device fulfills its core functional requirements and offers a good basis for further development. Compromises on the number of sensors and convenience features were made due to the thesis scope and shortage of storage space. Room for future optimization remains in the power consumption, wireless data throughput, and storage solutions.

Similar projects are suggested to focus on the flow of data in the software design, verify assumptions about the application early, and choose sensors and peripherals that are suited for low-power operation.

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Energy Systems
Sähkötekniikka

Harri Rahikainen

Puettavan datankeruujärjestelmän suunnittelu ja prototyyppi

Diplomityö

2021

55 sivua, 12 kuvaa, 6 taulukko(a) ja 0 liitettä

Tarkastajat: Professori Pertti Silventoinen
DI Antti Immonen

Hakusanat: Puettava elektroniikka, IoT, Anturi, Bluetooth Low Energy

Internetiin liitettävien laitteiden määrä kasvaa nopeasti. Esineiden internetin sovellukset kattavat monia yhteiskuntamme sektoreita älykkäästä infrastruktuurista teollisuuteen ja kulutuselektroniikkaan. Älykkäitä antureita käytetään ympäristön ja jopa ihmisen terveyden ja toiminnan mittaamiseen. Näiden älykkäiden laitteiden tuottama data on siirrettävä laitteilta internetiin prosessointia ja tarkastelua varten.

Tässä diplomityössä tutkittiin, miten voidaan suunnitella puettava tiedonkeruulaite, joka mittaa ja tallentaa itsenäisesti useita suureita sekä lähettää mitatun datan toiselle laitteelle. Ensin kerättiin ja käsiteltiin laitteen tekniset vaatimukset, minkä jälkeen toteutettiin prototyyppi.

Laiteprototyyppi toteutettiin sulautetulle mikrokontrollerille käyttäen reaaliaikakäyttöjärjestelmää. Työn tuloksena saatu prototyyppi mittaa sykettä, lämpötilaa ja suhteellista ilmankosteutta sekä tallentaa näiden suureiden arvot katoamattomasti. Prototyyppi käyttää Bluetooth Low Energy -teknologiaa tallennetun mittaus- ja tilatiedon luotettavaan lähettämiseen yhdyskäytävänä toimivalle laitteelle.

Toteutetun prototyypin voidaan todeta täyttävän keskeiset toiminnalliset vaatimuksensa ja tarjoavan hyvän pohjan jatkokehitykselle. Sensorien määrän ja ei-kriittisten ominaisuuksien suhteen tehtiin myönnytyksiä diplomityön laajuuden ja tallennustilan puutteen vuoksi. Jatkokehityksessä voitaisiin optimoida laitteen virrankulutusta, langatonta tiedonsiirtonopeutta sekä mittaustiedon tallennuksen ratkaisuja.

Samankaltaisia projekteja ohjeistetaan kiinnittämään huomiota tiedon kulkuun ohjelmistoa suunnitellessa, tarkistamaan sovellukseen liittyviä oletuksia aikaisessa vaiheessa sekä valitsemaan pienen virrankulutuksen toimintaan sopivia antureita ja oheislaitteita.

ACKNOWLEDGEMENTS

Thank you to the LUT Laboratory of Applied Electronics crew for the interesting and challenging topic. Special thanks to M.Sc. Antti Immonen and M.Sc. Jesse Tolvanen for the remote support and advice throughout this research process.

I am forever grateful to my parents, my sister Heini, Sofia, best of friends Jesse, Aleksi, Jami, Samuel and Venla, Sätky, the guild room, and the various food and refreshment establishments around the LUT campus for keeping me alive and sane during my studies.

On tämä kyllä yks työmaa.

Harri Rahikainen
Lappeenranta September 27, 2021

CONTENTS

ABSTRACT
TIIVISTELMÄ
ACKNOWLEDGEMENTS
CONTENTS
NOMENCLATURE
1 INTRODUCTION . . . 8
1.1 Background and Motivation . . . 8
1.2 Objectives and Scope . . . 9
1.3 Research Methods . . . 9
1.4 Structure . . . 9
2 SYSTEM REQUIREMENTS . . . 10
2.1 Functional Requirements . . . 11
2.2 Non-Functional Requirements . . . 12
2.3 Next Release Features and Constraints . . . 13
2.4 Requirements Rationale . . . 15
3 TECHNICAL BACKGROUND . . . 17
3.1 Bluetooth Low Energy . . . 17
3.2 Serialization . . . 22
4 DESIGN AND IMPLEMENTATION . . . 25
4.1 Choice of Development Tooling . . . 25
4.1.1 nRF5 SDK . . . 25
4.1.2 Nordic Connect SDK . . . 25
4.1.3 SDK Comparison . . . 26
4.2 System Architecture Overview . . . 27
4.3 Sensing . . . 29
4.4 Storing of Sensed Data . . . 33
4.5 Transmitting Stored Data to the Gateway . . . 38
5 ANALYSIS AND DISCUSSION . . . 46
5.1 Capabilities of the Prototype . . . 46
5.2 Review of the Prototype Requirements . . . 46

5.2.1 Review of the Functional Requirements 46
5.2.2 Review of the Non-Functional Requirements 48
5.3 Future Improvement Areas 49
5.4 Prototype Design and Implementation Key Takeaways 51
6 CONCLUSIONS 52
REFERENCES 53

NOMENCLATURE

Abbreviations

ADC	Analog-to-Digital Converter
API	Application Programming Interface
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
BPM	Beats Per Minute
CCCD	Client Characteristic Configuration Descriptor
DAQ	Data Acquisition Device
GAP	Generic Access Profile
GATT	Generic Attribute Profile
I2C	Inter-Integrated Circuit
IoT	Internet of Things
JSON	JavaScript Object Notation
L2CAP	Logical Link Control and Adaptation Protocol
LL	Link Layer
MTU	Maximum Transmission Unit
NCS	Nordic Connect Software Development Kit
NUS	Nordic UART Service
QSPI	Quad Serial Peripheral Interface
RTOS	Real-Time Operating System
SDK	Software Development Kit
SM	Security Manager
SoC	System on Chip
SPI	Serial Peripheral Interface
TWI	Two-Wire Interface
UUID	Universally Unique Identifier

1 INTRODUCTION

This chapter introduces the topic, objectives, and structure of the study.

1.1 Background and Motivation

The Internet of Things (IoT) has become a technological phenomenon in recent years. The number of connected devices is projected to reach 25 billion by the year 2030. The applications of IoT span many sectors ranging from smart cities and buildings to environmental and personal monitoring (Statista, 2020). Wearables such as smart watches, straps or earbuds are smart devices that can be worn on the body and are becoming an increasingly significant component of IoT. Wearable IoT can be divided into four primary categories: health, activity monitoring and sports, tracking and localization, and safety. The health category includes use cases such as patient monitoring, treatment and rehabilitation. Activity monitoring and sports include monitoring daily habits and athletic performance and using the data to improve health and results. Tracking applications focus on finding and following the position of humans or animals. Finally, wearables can be used to monitor the condition of the user and their environment for hazards (John Dian et al., 2020, pp. 69200-69205).

The commissioner of this thesis, Lappeenranta-Lahti University of Technology LUT (LUT University) Laboratory of Applied Electronics has participated in research around the health and activity monitoring of humans. The laboratory has had a recurring need for a mobile data acquisition platform in its research projects. When developing a new design for a sensor, researchers must test the sensor and get measurement data to the cloud for further processing and analysis. Previous projects have used improvised solutions that are not adaptable to future projects, utilizing, for example, the Raspberry Pi. As suggested by Nakamura et al. (2017), although beginner-friendly embedded system platforms have been emerging in recent years, it is not easy to succeed in developing small form-factor measuring devices from them. This thesis work was initiated to develop an easily extendable, well-documented data acquisition platform to avoid repetition and wasted efforts in the future.

1.2 Objectives and Scope

The objective of this thesis is to design and implement a mobile data acquisition device prototype and document the implementation for easy use and extensibility. The design challenges of a data acquisition platform include the mobile small form-factor hardware, embedded firmware, and user application development. The scope of the thesis is limited to the embedded device software component of the system leaving the hardware and processing of the data for further development.

The research questions for the study are formulated as follows:

- What are the requirements of the data acquisition platform?
- What would be a suitable design for a wearable data acquisition device that measures and stores multiple quantities and transmits the measured data?

1.3 Research Methods

- Gathering requirements regarding current and future project needs
- Formulating a specification based on requirements
- Designing and implementing the prototype
- Documenting and analyzing the implementation

1.4 Structure

Chapter 2 describes the requirements for the data acquisition platform, which were gathered and formulated with the client. In chapter 3, the key technologies are introduced. The prototype design and implementation are described in detail in chapter 4. In chapter 5, the prototype is analyzed and suggestions for improvements and future development are given. Finally, the conclusions are presented in chapter 6.

2 SYSTEM REQUIREMENTS

This chapter describes the purpose of the data acquisition system and the requirements of the data acquisition device prototype as a part of the system and as the main focus of this thesis.

The requirements were gathered via meetings and longer workshop sessions with researchers of LUT University Laboratory of Applied Electronics who had previously been working on or would be working on projects centered around data acquisition in the future. The laboratory's current and projected needs are in measuring biosignals such as heart rate and heat flux. Another common theme and requirement is mobility or wearability, in other words, wireless connectivity.

Through the collaboration with the client, a mutual understanding of the system was formed. The envisioned system consists of a data acquisition device, an Internet-connected gateway, and a cloud service. The data acquisition device measures and stores data independently without a reliable connection available at all times. The user of the system can download the stored data from the device using the gateway application. The gateway application then allows the user to view important metrics, annotate measurement results and upload them to the cloud service for storage and further analysis. The concept of the system is presented in Figure 1.

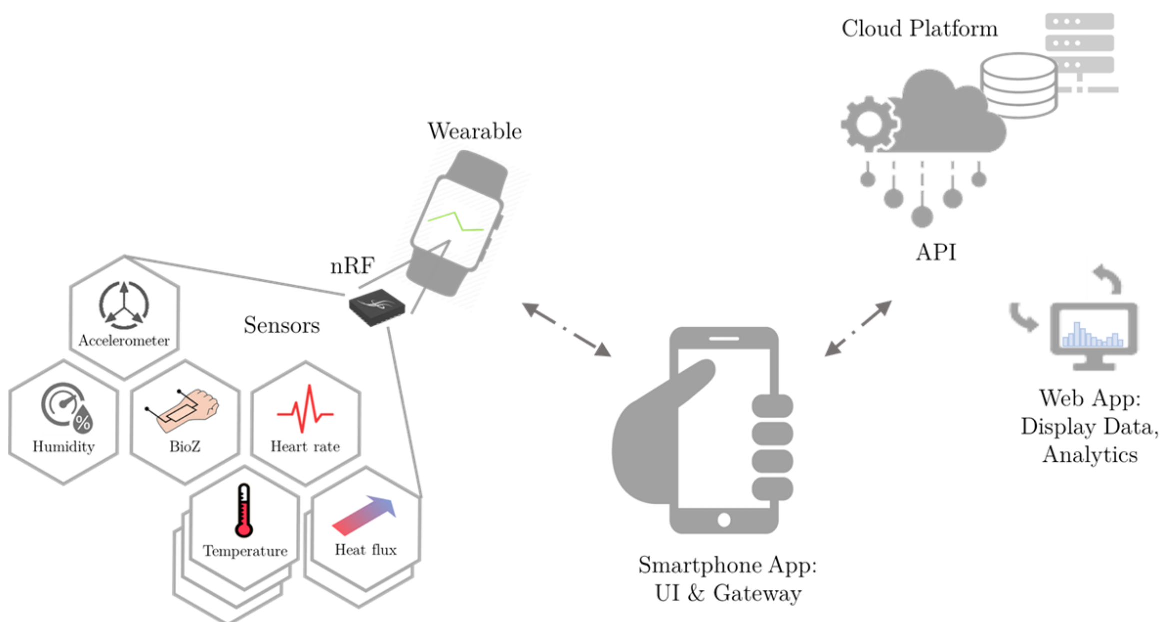

Figure 1. Envisioned data acquisition system including the wearable data acquisition device, mobile gateway application, and cloud services.

In the system depicted in Figure 1, the wearable data acquisition device (hereinafter referred to as DAQ) is a battery-powered, rechargeable module that can be attached to different parts of the human body, for example, to the wrist as a bracelet or to the waist as a belt clip. The DAQ samples signals from its attached sensors at their individual rates and saves the signals locally to its non-volatile memory. When the gateway device ("Gateway") is near the DAQ, the user of the system can form a wireless connection to the DAQ to download the local measurement data. The Gateway may be a smartphone, as shown in Figure 1, or another Internet-connected device with sufficient computing power, wireless connectivity, and display capabilities to provide the user the data handling and transmission services. From the Gateway, the data is eventually posted to the cloud platform ("Cloud"). The gateway is connected to the Internet and will communicate with the Cloud to post data and get analytics results back for display. The Cloud offers persistent storage of the posted data and allows querying and downloading it. The intended users are researchers. The Gateway and Cloud applications may be used by study participants or research project stakeholders for visualizing data.

2.1 Functional Requirements

Functional requirements describe what the system should do. The results of the requirements gathering for the DAQ have been distilled to the requirements presented in Table 1.

Table 1. Functional requirements of the DAQ. Each requirement is labeled uniquely and assigned a priority from 1 to 4 in descending order.

Label	Description	Priority
R-K1	DAQ measures biosignals and environmental variables with high quality.	1
R-K2	DAQ sends measurements to the connected gateway application reliably.	1
R-K3	While connected to the Gateway, the DAQ shall send the Gateway a notification if state changes, errors, or fault conditions occur.	1
R-K4	DAQ shall change measurement parameters, when it receives a command from the Gateway.	3
R-K5	While connected to the Gateway, the DAQ shall send the Gateway a notification when the battery charge level drops below 10 %.	3
R-K6	When not connected to the Gateway, the DAQ shall save measurement data to its non-volatile memory. If the data points don't fit into memory, the oldest data is overwritten.	1
R-K7	All data points from all sensors of the DAQ shall either include a timestamp or be traceable to a timestamp.	1
R-K8	If an error occurs, the DAQ shall be able to recover, for example, by rebooting or removing a faulty sensor from the measurement configuration.	1

The priorities in Table 1 are assigned in descending order. Priorities one and two are reserved for critical or "must-have" features which need to be included in the prototype for it to function as intended. Priorities three and four are treated as convenience or future development items which are implemented if there are enough resources left. The data quality, as mentioned in requirement R-K1, is clarified in Table 2 along with planned measurements and sensor types.

Table 2. Measured variables and their data quality requirements. Priorities are in descending order.

Prio	**Sensor type and quantity**	**Resolution**	**Accuracy**	**Sample rate [Hz]**	**Range/bits**
1	Optical Heart Rate	1 BPM	-	≥ 1	-
1	3-axis Acceleration	1 mg	-	25	±16g, 16 bits/axis
2	2×Temperature	0.001°C	0.1°C	0.1	10-45°C/ 24 bits
2	2×Relative Humidity	0.1 %	5 %	0.1	-
2	IR Temperature or Heat Flux	1 μV	100 μV	0.1	-10 - 10 mV / 24 bits
3	Bioimpedance	1 kΩ	50 kΩ	0.1	0-1 GΩ/ 16+ bits

The prototype hardware is planned to have dedicated sensors for measuring heart rate and acceleration. Different analog measurements, such as the temperature, heat flux, and bioimpedance presented in Table 2, can be connected through an analog-to-digital converter (ADC).

2.2 Non-Functional Requirements

Non-functional requirements describe how well the system does something or what qualities it exhibits. The qualities identified for the operation of the system and the DAQ itself are discussed below.

Portability

In this project, portability means that the Cloud can be deployed to different service providers such as Amazon Web Services and Microsoft Azure without a large effort. This will ease the transition from the development environment to LUT University's machines and any provider that they choose to use at a given time. In practice, this means designing the Cloud and its database connection so that they can be run on any Linux machine as Linux is ubiquitous in the cloud service provider space.

Learnability

Learnability means that a new user of the data acquisition system can set up and use the DAQ and Gateway with documented instructions. For research and further development, sufficient documentation and developer tools shall be provided.

Reliability

Reliability is defined in terms of hardware and data. The DAQ shall continue to operate and measure with high quality regardless of the temperature changing in the range of 0°C– 40°C or the voltage fluctuating between 3 V and 4.3 V. The DAQ shall be usable outside laboratory conditions, both indoors and outdoors.

The DAQ shall not lose long periods of data in transfer to the Gateway. A long period is defined here as one minute or longer. The DAQ shall also try to recover from disturbances caused by, for example, low battery charge or disconnection from Gateway without corruption of data. The data shall be saved to non-volatile memory to wait for recovery.

Usability and Comfortability

The DAQ can be worn by a user continuously for multiple days without feeling major discomfort. The minimum time period required by the client is two days. The DAQ shall not cause irritation of the skin, weigh down the bearer or otherwise inconvenience them. Basic functions including putting on and taking off the DAQ, and charging the DAQ shall be easy to use.

Battery Life

As the DAQ is wearable, it must operate on battery power. In this project, it was agreed that the battery should last for a minimum of 24 hours. Implementation is planned to achieve a 48-hour battery life and a future development aim is to increase the time between charges to a week.

2.3 Next Release Features and Constraints

The scope of the data acquisition platform prototype was constrained to fit into the thesis project timeline and some more advanced features were listed as nice-to-haves, meaning that they are not required for acceptance of the delivered prototype. These features were left for the future development of the platform. The focus of this thesis is solely on the DAQ software while the development of hardware, Gateway, and Cloud is ongoing in parallel by the rest of the team.

Features that will not be implemented in the prototype are listed below:

- Hardware modifications
- Advanced security
- Configuration at run-time
- Firmware updates

The development of the DAQ involves both software and hardware. The software and hardware are developed by different people inside the team. Hardware prototypes will continue to be developed throughout the project duration, but the base platform and components will remain the same during the thesis timeline. Therefore, possibilities for hardware-based optimizations are very limited and the software must be designed for the pre-established version of the hardware in this thesis.

Security is a complex topic both in software and hardware. In the context of the first prototype, security is addressed on a basic level. The Cloud shall have basic access control i.e. admin and normal users. The communication between the Gateway and the Cloud shall use token-based authentication of requests. On the embedded software side, encryption of wireless traffic is used.

Changing configuration is a convenience feature in the context of this prototype. The configuration parameters of the system could be static or changeable during the execution of the software. In the static case, multiple predefined sets of parameters could exist on the device. This would require greater memory usage. The software could alternatively provide the user a possibility to change the parameters in place during execution through an interface. This runtime configuration option is not implemented in this thesis in order to focus efforts on the core functionality.

In complex applications, it is often necessary to update the software after the device has been deployed to the field or when testing with the production form factor has been started. Updates can be delivered in multiple ways, both wired and wireless. Common examples include updates by UART, USB, Wi-Fi, and Bluetooth Low Energy. Wireless updates are sometimes called Over-the-Air updates. Wireless updates can be especially useful when the device is shielded or manufactured in a small form factor, making it hard to establish a wired connection. Firmware updates need to be secure so that devices do not execute new software from unknown sources.

There is also a danger of rendering the device useless if an update fails due to errors in the updating procedure itself. Due to this complexity, firmware updates are left for future development. The DAQ prototype hardware will include a wired interface to erase or reprogram the device but this interface is not used to update software while testing is ongoing.

2.4 Requirements Rationale

This section aims to elaborate on why the requirements are as described above.

R-K1 Data quality

The upcoming research at LUT University is focused on measuring human activities. Therefore, the highest priority sensor measurements, as seen in Table 2, were agreed to be heart rate and acceleration as well as temperature, humidity, and infrared temperature. Bioimpedance is left with a nice-to-have status for the first prototype and will be implemented in future versions.

Acceleration can be used for human fall and activity detection and measuring it requires a higher sample rate in order to detect fast events. Temperature and relative humidity values may be used to differentiate between outdoor and indoor conditions of activities and to serve as complementary information on a person's condition (John Dian et al., 2020, pp. 69205-69206).

R-K3 & R-K5 Status updates

From the perspectives of both the user and the developer, it is desirable to have visibility into the system. On a basic level, this means knowing that the system is on and the sensors are intact. Any status codes can be read on the Gateway and the troubleshooting will be easier. As a convenience feature, the DAQ shall notify about low battery. This is a common alert in mobile phones and smartwatches.

R-K4 Changing configuration

To enable rapid research and development, it would be convenient for the users to have the ability to change measurement parameters, e.g the gains of acquisition channels, sample rates, or filters while the software is executing. This could mean having predefined parameter sets for particular sensors which can be switched between on user command.

R-K6 Storing data

The DAQ shall have non-volatile storage for the measurement data. The wireless connection between the DAQ and the Gateway may and will fail at times, for example when the devices go out of range of each other or some obstacle weakens the signal. Furthermore, continuous wireless data streaming would increase power consumption and demand more bandwidth from the connection. Therefore, during these situations, the DAQ shall continue taking measurements and store them internally until sending them becomes possible again. If a connection cannot be found for some time and the DAQ runs out of non-volatile memory, the oldest measurements shall begin to be overwritten. This is a practical decision for the first prototype. The memory requirements will be re-evaluated in future versions of the system.

R-K7 Timestamping

It is crucial, that the measurements contain a timestamp of when they were taken. Time series data is stored in the cloud for later analysis and visualization.

R-K8 Error handling

It is important that error handling procedures are taken into consideration in the design of the software. For operations that may fail because of hardware-related issues, proper error code checking and retry mechanisms should be put into place. Even if there is no way for the system to recover from an error state, for the purposes of further analysis and development, it is useful to send an error message by some form of telemetry or store the error and any events leading up to it in non-volatile memory for later retrieval by maintenance personnel. To facilitate problem-solving while moving to a smaller form factor, it can be useful to leave a debugging interface in the early prototypes so that logs or other debug information can be accessed in the testing phase.

Battery life

Battery life should be minimum of 24 hours and up to 48 hours in order to enable measuring human night-time activities and a whole day cycle without removing the device for recharging. It is also more convenient for the user if they do not need to charge the device as often.

3 TECHNICAL BACKGROUND

This chapter introduces key concepts of the technology in the data acquisition system.

3.1 Bluetooth Low Energy

Bluetooth is a wireless communications system that uses the 2.4 GHz frequency band. The Bluetooth standards are developed and managed by the Bluetooth Special Interest Group (SIG). The Bluetooth Core Specification includes two forms of the technology: Basic Rate (BR) and Low Energy (LE). Low Energy was added in version 4.0 of the Bluetooth Specification and is widely referred to as Bluetooth Low Energy (BLE). Bluetooth Low Energy was designed for low power consumption, low data rate, and low-cost use cases. Basic Rate applications include various cable replacement use cases such as hands-free headsets, whereas BLE is widely used in wireless sensor applications (Bluetooth SIG, 2021, p. 187).

The Bluetooth system consists of functional, conceptual blocks and their associated protocols. These architectural blocks and protocols form layers which are further grouped into the Host and the Controller.

The Controller consists of the two lowest layers, the LE Radio and Link Layer (LL). The Host is comprised of the Logical Link Control and Adaptation Protocol (L2CAP), Attribute Protocol (ATT), Security Manager (SM), Generic Attribute Profile (GATT), and Generic Access Profile (GAP), listing from the bottom up. The Host Controller Interface (HCI) lies between the Host and the Controller to enable interoperability between implementations (Bluetooth SIG, 2021, p. 202-206). The protocol stack layers are introduced in more detail below.

The LE Radio is the physical layer (PHY), which handles the analog communication over the air (Townsend et al., 2014, p. 16). The maximum symbol rates are 1 Mbps and 2 Mbps for Bluetooth versions 4 and 5 respectively with the 2M symbol rate being optional (Bluetooth SIG, 2021, p. 190).

The Link Layer is a high-performance part of the Bluetooth protocol stack and is responsible for managing the link state of the radio. It interfaces with the below physical layer and the L2CAP layer above. Tasks of the Link Layer include, among others, advertising, scanning, the creation and teardown of connections, transmitting and receiving packets, encryption and

decryption of data, and connection parameter updates. The Link Layer uses the terminology Advertiser-Scanner and Master-Slave for the interacting parties while outside and in a connection (Townsend et al., 2014, pp. 16-24).

L2CAP multiplexes the upper and lower layer packets in either direction. It breaks down the upper layer protocol packets to suit the LL packet format and conversely combines multiple received LL packets for the upper layers (Townsend et al., 2014, p. 25).

The ATT and SM protocols lay on top of the L2CAP. The ATT protocol defines Client and Server roles for exchanging messages. The Server stores data in structures called attributes. The attributes can be identified with Universally Unique Identifiers (UUID). The ATT Client may send commands, request information, or send confirmations to the Server. The ATT Server then sends responses to the Client (Bluetooth SIG, 2021, p. 279). The Bluetooth security model uses security keys to verify device identities and encrypt communications. The Security Manager defines methods and the protocol for pairing and security key distribution and cryptographic algorithms to support them (Bluetooth SIG, 2021, pp. 1554-1556).

The GATT is built on top of the ATT and uses the underlying ATT procedures to perform operations and exchange data. It defines a data model where the ATT attributes are grouped in a hierarchy of profiles, services, and characteristics. Services contain one or more characteristics. Characteristics include a characteristic value, properties such as read and write permissions, and possible descriptors for that value data such as units or scaling factors. GATT profiles are the highest level of the hierarchy and organize services for a use case so that interoperability between compliant implementers is possible (Bluetooth SIG, 2021, p. 280).

The GAP is a mandatory profile common for all BLE devices, describing device roles and the behaviors and methods for discovering devices, establishing connections, security and authentication, association models, and discovering services. In this case, the profile is used as a term to describe a vertical slice of the Bluetooth stack layers such that devices are able to interact in a consistent manner (Bluetooth SIG, 2021, pp. 277-278).

Bluetooth Low Energy devices can send out advertising packets to communicate and announce their existence. Communication between BLE devices can happen entirely through broadcasting advertising packets or they can alternatively send connectable advertising packets (Bluetooth SIG, 2021, p. 191). Advertising packets can hold a very limited amount of data payload and

the broadcasted data is open to any observer. Furthermore, there is no guarantee that a specific observer will be nearby to receive an advertising packet (Townsend et al., 2014, p. 19).

Devices that receive connectable advertising packets may send a connection request to the advertiser. Upon the advertiser accepting the request, a connection is established between the devices. The parties in the connection are referred to as Central (initiator of the connection) and Peripheral (the former advertiser). Within a connection, the Central and Peripheral can exchange data packets back and forth in time slots called connection events (Bluetooth SIG, 2021, p. 191).

The BLE connection is affected by the following connection parameters: connection interval, slave latency, and supervision timeout. Connection interval is the time period between connection events, ranging between 7.5 milliseconds at the minimum to a maximum of 4 seconds. Slave latency signifies the number of connection events that the Slave device is allowed to miss before the Master considers the connection lost. Supervision timeout is the maximum time between receiving valid packets and is monitored by both parties of the connection for detecting connection loss (Townsend et al., 2014, p. 23). The connection parameters are set by the Central device once the connection has been established. The Central may use parameters preferred by the Peripheral or ignore them and set different parameter values. The Peripheral may request an update to the parameters later in the connection. If the set or requested parameters are unacceptable to either party, they may choose to disconnect. The Bluetooth specification suggests that the devices should be flexible in accepting requested parameters (Bluetooth SIG, 2021, pp. 1324-1327).

In contrast to broadcasting-based communication, BLE connections allow varying degrees of security. The BLE system provides security features of data encryption, authentication, and privacy. The data over the BLE connection can be encrypted. The source of data may be verified by using data signing. Finally, the address of the device can be hidden from tracking by the private address feature. The protocol and algorithms for the security key generation and distribution are provided by the Security Manager. The method of exchanging the security keys depends on the input-output capabilities of the devices such as the availability of a display, keyboard, or buttons (Bluetooth SIG, 2021, pp. 274-275).

The data exchange between BLE devices is defined by GATT. The GATT, like ATT, defines the Client and Server roles. The roles are not fixed to which device initiated the BLE connection, but defined when a procedure is initiated. Both Central and Peripheral devices can act as GATT Client and Server. When the connection is first established, the GATT Client is unaware of the GATT services and characteristics on the other device, the GATT Server. It must therefore first perform service discovery before any operations on the Server's attributes. The services and characteristics are identified by their UUIDs. UUIDs may be either standard or vendor-specific, in other words, custom generated by the application developer (Townsend et al., 2014, pp. 51-54). After the Client knows the available services and characteristics on the Server, it has five main options to exchange the characteristic value data: Read, Write, Write Without Response, Characteristic Value Notification, and Characteristic Value Indication. The last two are initiated by the GATT Server once they have been enabled by writing to a Client Characteristic Configuration Descriptor (CCCD). The Server can send a notification or an indication when the value has been updated, for example, when a new measurement has been taken. Notifications and indications allow for more efficient communication as the Server does not need to be polled for information. The difference between notifications and indications is that indications are acknowledged with a confirmation from the Client whereas notifications are unacknowledged. Write Without Response is an unacknowledged write operation from the Client (Townsend et al., 2014, pp. 68-72).

The amount of user data that can be exchanged in a given time, the throughput, is dependent on a number of factors. First, the size of the user payload in a BLE air packet depends on the Bluetooth version that the implementation supports. Before version 4.2, the maximum LL payload was 27 bytes including header overhead from the upper layers. From the Bluetooth specification 4.2 onward, the limit is now 251 bytes (Bluetooth SIG, 2021, p. 2700). This increased payload feature is optional and can be enabled with a procedure named Data Length Update, in which the devices exchange the maximum packet sizes they are able to transmit and receive (Bluetooth SIG, 2021, p. 2860). The larger the user payload per physical BLE packet the less overhead time is used.

Second, the used GATT operation affects the overhead and therefore the effective user payload as the ATT operation code and targeted attribute handle are transmitted in the GATT header. For writes, notifications, and indications, this overhead is three bytes per ATT packet. The

maximum amount of data that can be exchanged in a single ATT transaction is called the Maximum Transmission Unit (MTU) (Bluetooth SIG, 2021, p. 1415). The MTU size can be exchanged between devices. This procedure simply notifies the other device that the initiator is able to handle more data than the default 23 bytes of ATT payload. After the exchange, the MTU is set to the capacity that both devices can support (Bluetooth SIG, 2021, p. 1497). Larger MTU results in fewer ATT packets per transaction, which reduces the overhead from the operation headers. The L2CAP handles the segmentation of the ATT packets to LL payloads.

Third, the used connection interval is important. Acknowledged ATT operations, for example, read, write, and indication, require a response or confirmation which is sent in the following connection event. Therefore, acknowledged transactions take at least two connection intervals to complete. In contrast, unacknowledged operations, write without response or notification, can be sent at any time during a connection event. Given the nature of acknowledged transactions, the amount of user data that may be transmitted in two connection intervals is then MTU minus the GATT header. For unacknowledged operations, multiple ATT packets can be sent within a connection event (Bluetooth SIG, 2021, p. 1420). However, the packets need to be transmitted completely within an event. Therefore, the number of packets in an interval should be balanced so that the leftover time for the next connection event is minimal. It should also be noted that with a given symbol rate, larger payloads naturally take longer to transmit. In all operations, if errors are encountered during transmission, the packet is retried only in the next connection event. Thus, a longer connection interval may cause wasted time in an environment with a lot of interference (Gomez et al., 2012, pp. 11747-11748). Finally, the used Bluetooth protocol stack implementation may limit the payload size further (Townsend et al., 2014, p. 7).

As the name suggests, Bluetooth Low Energy was designed with power consumption in mind. BLE Central devices may often have less strict battery life requirements since they might be connected to a power source like USB BLE adapters or charged often like smartphones. The power consumption of BLE Peripheral devices, such as the DAQ in this study, is more critical to optimize as they are more commonly designed to operate for extended periods of time, for example, on a coin cell battery (Gomez et al., 2012, p. 11741). The operation of Peripheral devices is characterized by short bursts of transmitting activity and deep low-power mode sleep in between them (Townsend et al., 2014, p. 8).

In Bluetooth Low Energy advertising, the power consumption can be affected by adjusting the advertising interval i.e. how often an advertising packet is broadcast (Townsend et al., 2014, p. 19). Since advertising packets are used for forming connections, the advertising interval also affects the responsiveness of the BLE peripheral device. A longer interval means the product is less responsive, especially since connecting to the device might take the scanning Central device more than one advertising interval because packets can get lost or more data may be requested.

At the physical layer, the transmit power and modulation rate affect consumption. Higher transmit power enables a longer range but comes at the cost of battery life. A longer range may be more convenient for the user of the BLE device so there is a trade-off between user experience and lower power consumption (Townsend et al., 2014, p. 8). With Bluetooth 5 and its 2 Mbps bit rate, power consumption gains are available for the devices that support the new modulation scheme. A faster modulation rate means that the data is sent in less time and the radio is active for a shorter period, as demonstrated by Bulić et al. (2019).

In BLE connections, the power consumption is influenced by the connection parameters and the used operation. The connection interval is adjustable between 7.5 milliseconds and 4 seconds to allow for different needs between throughput, responsiveness, and lower power. A larger connection interval value allows the devices to sleep for longer and provides a deterministic schedule of when to wake up (Townsend et al., 2014, pp. 8-12). Gomez et al. (2012) emphasize that slave latency has similar benefits for Peripheral devices that do not need quick response times to requests. The efficiency of the BLE communication can also be increased by using notifications or indications from the GATT Server instead of request-based polling from the Client.

3.2 Serialization

There are standard services defined by the Bluetooth SIG with reserved UUIDs. Some of these services lend themselves well to wearable data acquisition applications. For example, Health Thermometer, Battery Service, and Heart Rate Service are very useful when measuring biosignals and operating on a battery-powered device.

For parameter changes such as sampling rates, voltage gains, offsets, or calculation thresholds, creating new services and characteristics might become cumbersome and repetitive fast. Similarly, for data streaming purposes, one might end up having a different characteristic for

each type of measurement. The complexity of adding new services and characteristics also grows on the central application side, where the application has to filter through UUIDs of each desired characteristic.

An alternative approach to using standard Bluetooth services and separate characteristics could be to use only one service and characteristic as the communication channel. This characteristic could hold a variable-length value up to a predefined maximum length. The value would then be used to send different messages containing data, commands, and responses. This would move the complexity toward a custom protocol or data format with BLE as the transport. The programmer would be free to implement any data format and make sure that the Central and Peripheral encode and decode the messages in the same way. In the development of the DAQ, this trade-off between many services and characteristics and a custom data format was explored.

Some vendors choose to implement a service that emulates a serial port over Bluetooth. In the versions before Bluetooth 4.0, a Serial Port Profile (SPP) exists in the standard, but for BLE it needs to be defined and implemented separately. For example, Nordic Semiconductor has the Nordic UART Service (NUS) and Silicon Laboratories has examples of SPP over BLE. Through these, it is possible to send custom message payloads.

Since the Bluetooth transport handles data as a sequence of bytes or characters, the application data structures must be converted to and from this raw format. These processes are called serialization and deserialization. For embedded applications, these processes should be fast and memory-efficient and the resulting message sizes small in order to save bandwidth (Hamerski et al., 2018, p. 773). As Hamida et al. (2015) suggest, it is important to choose the serialization format according to the system's needs. If interoperability and extensibility are important, an open, standardized format would be preferred. On the other hand, they note, a custom proprietary format allows for more control to the implementer. Other factors that affect the choice of serialization format and the performance of the messaging are whether the data is compressed before or after the serialization, and the quality of the serialization library. The library choice should consider not only the performance but also the development activity and support (Petersen et al., 2017). Several candidates for the data format are introduced below and the final choice for the DAQ is presented in chapter 4.

JSON (JavaScript Object Notation) is a textual and programming language-independent data interchange syntax that was derived from the JavaScript programming language. JSON is simple, human-readable, and implementable through language constructs of key-value pairs and arrays (Ecma International, 2017, p. 5). Commonly used in web applications, JSON provides the benefits of readability and interoperability between languages but results in a larger message size than for example binary formats (Hamerski et al., 2018, p. 774).

Protocol Buffers or protobuf, are a data interchange format developed by Google. Protocol Buffers allow the user to define the data as structured messages in the protobuf syntax and generate encoding and decoding source code for multiple programming languages using the provided tooling. The messages are sent in binary format (Google Inc., 2021). Performant implementations targeted at memory-constrained devices exist, for example, *nanopb* (Aimonen, 2021).

MessagePack is a binary serialization format which claims to be faster and smaller than JSON. It has implementations in various programming languages and environments (Furuhashi, 2019). Hamerski et al. (2018) found several MessagePack libraries that performed well in an embedded controller setting.

The Concise Binary Object Representation (CBOR) is a data format designed to be extensible and small both in code and message size. CBOR is defined in an Internet standard RFC 8949 and has implementations in many programming languages, such as TinyCBOR for C and C++. CBOR's data model is based on JSON and CBOR operates without a schema for the data (Bormann, 2020).

4 DESIGN AND IMPLEMENTATION

In this chapter, the design decisions and the resulting implementation of the prototype are presented.

4.1 Choice of Development Tooling

The hardware platform the prototype is developed on is the Nordic Semiconductor nRF52840 System on Chip (SoC), which has an ARM Cortex-M4F 64 MHz core with a floating-point unit, 1 MB of internal flash memory, 256 kB of RAM, and a +8 dBm transmit power capable BLE 2.4 GHz radio (Nordic Semiconductor, 2021c). Nordic Semiconductor currently offers two Software Development Kits (SDK) for customers to use when building products on their platform: the nRF5 SDK and the Nordic Connect SDK (NCS). This section compares the two SDKs and one is chosen for the development of software for the prototype.

4.1.1 nRF5 SDK

The nRF5 SDK is the older of the two SDKs. It supports the nRF51 and nRF52 series of hardware. It features libraries, peripheral drivers, examples, and protocol stacks. The Bluetooth Low Energy and other wireless protocol stacks are distributed in binary form and called SoftDevices. nRF5 SDK has mature support for the nRF52840 SoC with full support in version 15.0.0 in April 2018 and the latest version 17.0.2 released in September 2020 (Nordic Semiconductor, 2018). SoftDevices have been in development since 2011 and are also mature. The nRF5 SDK will not get new Bluetooth features after Bluetooth 5.0 specification (Kvaale, 2020, pp. 32-35).

4.1.2 Nordic Connect SDK

The Nordic Connect SDK is more recent with version 1.0.0 in April 2020, nRF52840 support in version 1.3.0 in July 2020, and the latest version 1.6.0 released in June 2021 (Nordic Semiconductor, 2021d). It is a collection of open-source projects maintained by Nordic Semiconductor. The NCS includes the Zephyr™ real-time operating system (RTOS). In addition to the RTOS, NCS provides drivers, libraries, and examples specific to Nordic Semiconductor hardware. Some libraries are provided as closed-source binaries similarly to the SoftDevices. The NCS supports development on nRF52 and the newer nRF53 and nRF91 series (Nordic Semiconductor, 2021a).

4.1.3 SDK Comparison

During the initial phase of the project, both SDKs were evaluated. Key information about the SDKs is compiled in Table 3.

Table 3. Software Development Kit information in short.

SDK property \ SDK	nRF5	nRF Connect
Supported hardware	nRF51 nRF52	nRF52 nRF53 nRF91
nRF52840 support released	04/2018	07/2020
Latest release	09/2020	06/2021
Supported Bluetooth specification	5.1	5.1+
RTOS	FreeRTOS/Any	Zephyr

Both of the SDKs have comprehensive documentation and examples ranging from basic to full applications. In both cases, it is simple to start developing one's own custom application on top of these existing samples. Nordic Semiconductor supports both SDKs through its online support forum called DevZone.

Both SDKs include libraries to accelerate development. In NCS, the developer is able to leverage the various subsystems of Zephyr related to for example storage and settings and configuration, in addition to wireless protocol-specific modules, which both SDKs provide. The nRF5 SDK also includes the FreeRTOS™ real-time operating system kernel as an optional component but it is featured in only a few examples and is not the primary way to build applications with the SDK. Although the prototype application has few features at this stage, Zephyr's libraries and software subsystems may be useful for future development. Zephyr includes other supported hardware and additional networking stacks, it also gives the flexibility to consider alternative technical options in the future. While more features can bring complexity to the system, during the testing of NCS, the learning curve was not seen as a major hurdle and the benefits outweighed the initial complexity.

A benefit of NCS is that it is in active development. New features are being added to both NCS and the Zephyr RTOS. NCS also supports Nordic Semiconductor's new nRF53, nRF91, and future hardware series, as can be seen from Table 3. Zephyr is an open-source project and its Bluetooth Low Energy implementation and other protocol stacks are thus also open.

In NCS, Nordic Semiconductor provides its own BLE controller implementation that can be used instead of the Zephyr counterpart. The Zephyr Project is neutrally governed, which is desirable in order to avoid being locked to a single vendor's way of operating (Zephyr Project, 2020c). Using Zephyr's interfaces, it is possible to develop software in a portable way that is not tightly coupled to a single platform.

Nordic Semiconductor provides the following guidelines on its website:

> "For nRF52 Series the nRF5 SDK and SoftDevices is still a valid option, if you are familiar with it and don't require an RTOS or features added after Bluetooth 5.0. If not, nRF Connect SDK is the correct choice." (Nordic Semiconductor, 2021b)

Considering all of the above, the Nordic Connect SDK was ultimately chosen for the development of the prototype software. Although the nRF5 SDK is solid and proven, the added features and future extensibility of NCS were considered to justify the usage of the new platform.

4.2 System Architecture Overview

This section provides an overview of the prototype system architecture and significant components.

Based on the requirements, the primary functions of the system are: measuring quantities with its sensors, storing the measured data on the device, and transmitting the stored data to a Gateway device for further processing. The system is therefore designed to be composed of subsystems that fulfill each function respectively. The high-level data flow in the system is illustrated in Figure 2.

Figure 2. Application data flow. The data sampled by the sensors is stored in the memory on the device and transmitted via Bluetooth when a connection becomes available.

As shown in Figure 2, the data is sampled by the sensors and then buffered and saved into non-volatile memory. Finally, when a Gateway device connects to the system via Bluetooth

Low Energy protocol, the stored data is loaded from the persistent memory and sent in encoded batches of data points to the Gateway.

In this first prototype hardware, there are three operational sensors: one optical heart rate sensor and two instances of a temperature and relative humidity sensor. The nRF52840 SoC and Bluetooth antenna come in an integrated, certified module package. The memory device on the hardware board is an 8 MB NOR flash memory, accessible through Serial Peripheral Interface (SPI). In the nRF52840 controller, a dedicated peripheral is used to communicate with the memory with up to four data lines ("Quad SPI"). The system is powered by a 3.7 V Lithium-ion battery. The hardware block diagram of the system is presented in Figure 3.

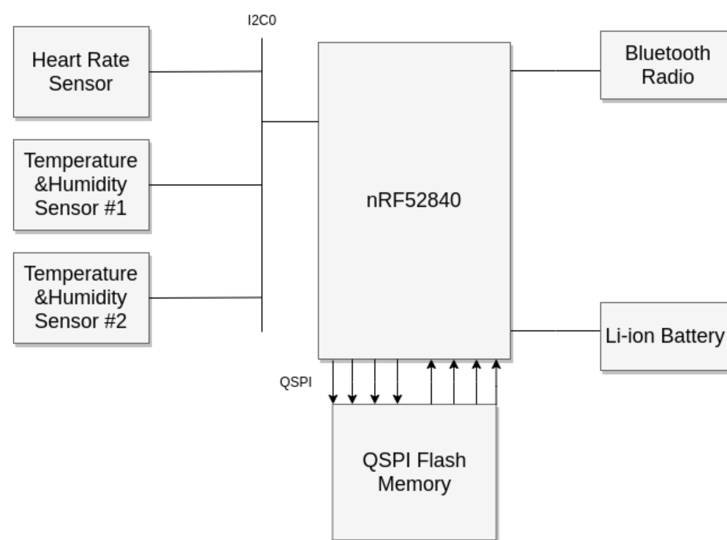

Figure 3. Hardware components of the system. The sensors are attached to the I2C0 peripheral and the Flash memory to the QSPI peripheral. The system is powered by a Lithium-ion battery.

As can be seen from Figure 3, all sensors are currently using the same I2C bus, I2C0. The two instances of the temperature sensor are configured to different I2C addresses with a hardware pin. The sampling rates of the sensors in the bus are 1 Hz for the heart rate sensor and 0.1 Hz for the temperature and humidity sensors, which are low and manageable with the same peripheral. The block diagram shows the Bluetooth Radio as a separate entity for simplicity, but in reality, the radio transceiver is integrated into the controller and connected to the antenna on the printed circuit board.

The software consists of different layers that each depend on the interfaces of the layer below them. By creating layers of abstraction and depending on an interface instead of an

implementation, the software can be made more reusable and portable (Beningo, 2017, pp. 18-20). The layered architecture of the system is presented in Figure 4.

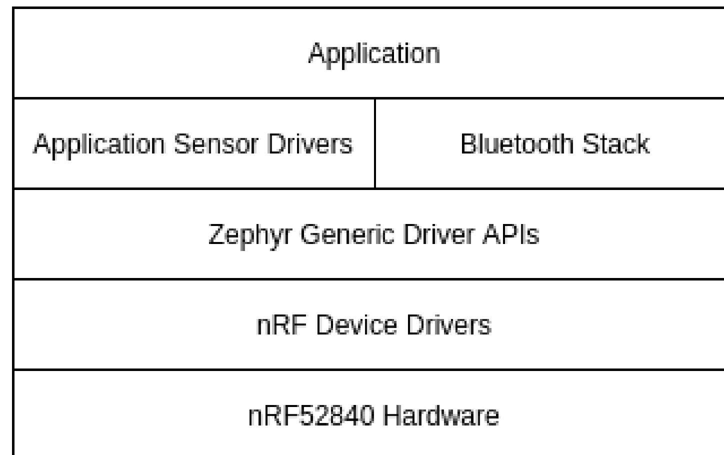

Figure 4. Software layers of the firmware. The application logic sits on top of drivers and modules that are generic at the top and get increasingly specific and concrete at the bottom.

The higher software layers in Figure 4 use the interfaces of the lower layers. At the bottom of the stack is the nRF52840 hardware. Above that are the device driver implementations for the specific peripherals of the Nordic hardware. Zephyr's driver interfaces abstract the platform away and refer to devices in a generic manner. For example, instead of "Nordic Two-Wire Interface", Zephyr uses "I2C". The application-specific sensor drivers use this generic I2C interface to interact with the devices. Finally, the user application can be programmed at a high level dealing in sensor values, files, and Bluetooth messages instead of the details of bytes and registers of the devices directly.

The following sections discuss the details of the implementation of each primary function.

4.3 Sensing

The prototype currently includes three sensors: one optical heart rate sensor and two instances of a temperature and relative humidity sensor. All three communicate through the I2C bus. There are plans to add software support for an accelerometer and an ADC in the future. These devices would be connected through the SPI bus. The hardware and software solutions to measuring with the sensors are described in this section.

First, the sensors need to be allocated to the buses of the nRF52 controller. The nRF52840 SoC shares peripheral resources, such as their state and configuration registers, in a way that

affects instantiation of I2C and SPI peripherals. Peripherals that share the same base memory address may share resources and their operation may be mutually exclusive at a given moment. If one switches between such peripherals, care must be taken to explicitly disable one peripheral, reconfigure and enable another peripheral (Nordic Semiconductor, 2021c, p. 98).

The nRF52840 features four SPI peripherals (SPI0-SPI3), two two-wire (I2C) peripherals (TWI0, TWI1), and a QSPI peripheral. Out of these, SPI0, SPI1, TWI0, and TWI1 share the same base address and thus cannot be used simultaneously. The user must select either SPI0 or TWI0 and either SPI1 and TWI1, respectively. SPI2 does not share resources with any other peripheral. SPI3 includes direct memory access (DMA) support, which is not necessary for this application nor necessarily portable. The QSPI is a dedicated peripheral and can be utilized solely to communicate with the external flash memory in this application.

The choice was made to use TWI0 to interface to the two temperature and humidity sensors and the heart rate sensor, since their sample rates are low (0.1 - 1 Hz). SPI2 is reserved to be utilized between the accelerometer and ADC in the future. These two have very different sample rates at 25 Hz and 0.1 Hz respectively and use different SPI modes. The accelerometer uses mode 0 and the ADC mode 3. There may be a future addition of another SPI mode 0 ADC. SPI1 and SPI3 are left disabled, but in case there is a need to sample the accelerometer or ADC at a faster rate and therefore to switch between the two SPI mode configurations more frequently, dedicating either one to the device with the highest sampling rate or differing SPI mode, in that case, could be considered. Leaving peripherals disabled when unused lowers the power consumption.

The software driving the sensors was developed following the layering shown in Figure 4. Configuring and sampling the sensors is done through interacting with their registers. The reading and writing of the registers rely on I2C. A driver for each sensor was developed in the C programming language. The driver software modules depend on the I2C interface of the given platform. The I2C bus read and write procedures are given to the driver module as C language function pointer arguments. The drivers are thus portable on top of different I2C implementations. In the prototype, the interfaces are provided by Zephyr.

The sensor drivers were designed to support multiple instances of a sensor on the same bus. For example, the temperature and humidity sensor in the system has a configurable I2C bus address. The address is configured in hardware, either pulling the address pin up to supply

voltage or connecting it to the ground. The prototype hardware has two of these sensors, so the sensor driver software must support different addresses for this sensor. Both sensors also have their own interrupt pins to notify the processor that data is available. These and possible differences in measurement parameters were taken into account in the design by including a pointer to a configuration structure as an argument to the initialization function of the sensor. On the application level, two separate configurations are defined and used to initialize two separate instances of a sensor structure. The sensor structure is then the first argument to all interfaces that the sensor driver provides. When the sensor driver needs to communicate via the I2C bus, it will use the address defined by the configuration linked to that sensor structure. The initialization of a temperature sensor instance is shown in Figure 5.

```
hdc2010_0_cfg = hdc2010_get_default_config();
hdc2010_0_cfg.addr                  = CS_PULL_UP;
hdc2010_0_cfg.int_pin               = 40;
hdc2010_0_cfg.sampling_rate         = TEN_SECONDS;

int hdc_0_err = hdc2010_init(&hdc2010_0, &hdc2010_0_cfg);
```

Figure 5. Temperature and humidity sensor initialization. The first of the two instances of this sensor has its I2C address pin pulled high. The init function is called with the specific instance and configuration.

In the above Figure 5, the first sensor's configuration structure denoted by index zero is set to a default configuration. The I2C address is then set to differ from the default with the *addr* field. The data interrupt pin is set to pin number 40 on the circuit board and the sampling rate is set to 0.1 Hz with a preprocessor alias *TEN_SECONDS*. Finally, the created configuration structure is passed as the second argument to the initialization function *hdc2010_init*.

The devices connected to the SPI bus, the accelerometer and the ADC, also have some driver software considerations. Devices may use different transfer speeds and SPI modes and they have their associated *chip select* pins that need to be managed. In the drivers themselves, the SPI bus transfer procedures are passed in as function pointer arguments so that the driver code stays decoupled from the particular hardware platform that it is used on. For the SPI transfers, different configurations of bit rate and mode are predefined for both devices' needs. These configurations are used by the functions that are passed into the drivers. When accessing these devices, the previous configuration is properly uninitialized before switching to another.

The application interacts with the sensor through a high-level interface. The interface provides functions to initialize the sensors, start and stop sampling, and register a callback function. The initialization function prepares and configures the sensors through the developed driver modules and starts the sampling procedure at the interval defined in the interface header file. The periodic sampling is achieved through a mechanism in Zephyr called Workqueue. Workqueues can hold *work items* that need to be executed. A work item is essentially a user-defined function to call when the item is processed. As the name *queue* suggests, the items are processed in order of submission to the queue. The items can be submitted to the queue instantly or with a defined delay (Zephyr Project, 2020b). The sensor sampling is done by using this delay mechanism to submit the next sample reading after the sampling interval. In this manner, there is no need to create a dedicated operating system thread only to wait on the sensor and thus memory resources are conserved. The workqueue mechanism is illustrated in Figure 6.

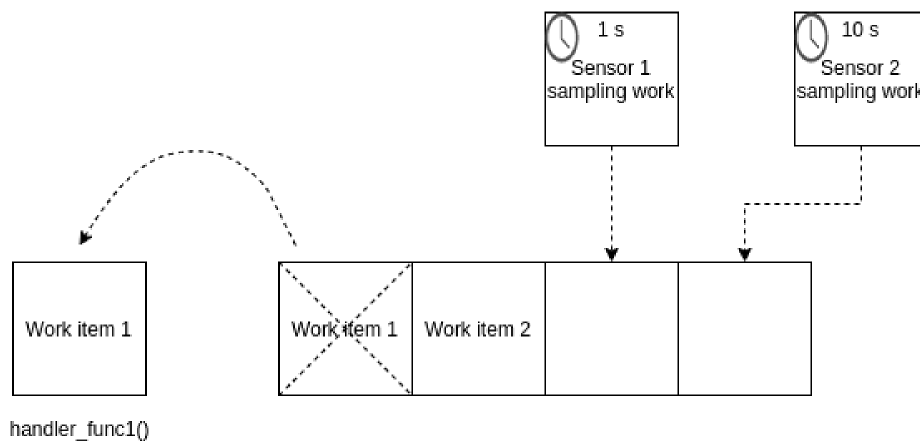

Figure 6. Zephyr RTOS workqueue basic principle. Work items are removed from the queue and their handler functions executed. Delayed work items are added to the queue after their specified intervals have passed.

The registered callback function, depicted as *handler_func1* in Figure 6, is executed when new data has been sampled. The callback function receives a copy of the sampled data along with a timestamp of when the sample was taken. The timestamp indicates the milliseconds that have passed since the last reboot of the system. The data from the heart rate sensor includes the heart rate in beats per minute (BPM). The temperature and humidity sensors include the raw readings of temperature and relative humidity as well as the sensor id number. The temperature and humidity module provides conversion utilities for transforming the raw readings to floating

point degrees Celsius and humidity percentage. The application uses the initialization and start functions at bootup. The callback provided data is buffered for saving and sending.

4.4 Storing of Sensed Data

The non-volatile memory device used on the prototype board is a Macronix MX25R6435F 64 megabit serial NOR flash memory. It is accessible through QSPI and features both high performance and low power modes. It claims to have a data retention of 20 years and a minimum erase/program cycle count of 100 000 (Macronix International Co., Ltd., 2020). The memory is writable by byte, word, or 256-byte page and erasable as a whole or in blocks of 4, 32, or 64 kilobytes. This Macronix memory chip is also used on the nRF52840 development kit from Nordic Semiconductor.

As discussed in chapter 2, the LUT measurement interests are in human biosignals. The sensors in the prototype include the heart rate and temperature-humidity sensors. Further development goals are to include an accelerometer and an ADC for analog sensor measurements, which were left outside of the scope at this stage. Based on the sensors on the prototype board, the data items to be stored persistently were determined. The data items to be saved for each of the metrics are listed in Table 4.

Table 4. Data fields to be saved from each sensor. The third column indicates the data type that each field is stored as in the C language.

Sensor	Data fields	C Data Types
Heart Rate	Heart Rate BPM	UINT16
	Signal Quality [0-100]	UINT8
	Total Step Count	UINT16
Temperature&Humidity	Temperature Raw Reading	UINT16
	Humidity Raw Reading	UINT16
	Sensor ID	UINT8
Accelerometer	Acceleration Raw X	INT16
	Acceleration Raw Y	INT16
	Acceleration Raw Z	INT16
	Scale	UINT8
ADC	Channel ID	UINT8
	Channel Value Raw	UINT32
	Channel Gain	UINT8

The third column in Table 4 shows the C language data types of all the fields to be stored. The sizes are in fixed-width integer types. The abbreviations *UINT* and *INT* refer to unsigned and

signed integer values respectively. The numbers refer to the number of bits used to represent the value: 8, 16, 32, or 64. The data types can be used to calculate the memory space needed to store an individual sample. According to the table, for example, the size of temperature and humidity samples in the flash memory would be five bytes.

After defining the data items to be stored, the flash memory storage format was to be decided. The system should have a way to trace back the timestamp of a stored and loaded sample when the sample needs to be processed. The flash memory space should also be conserved in order to require as few offloading sessions as possible. This supports the use cases of measuring nighttime body conditions or long exercise sessions without interruptions within the limits of the battery life.

Perhaps the simplest manner of storing data points to the flash memory could be including an individual timestamp to each sensor measurement value. This would make every value traceable without a doubt, thus fulfilling requirement R-K7. On the other hand, the timestamp would impose some overhead, being a 64-bit value in the case of Zephyr. This overhead might be a significant proportion of the total saved record for an individual data point. For example, a 3-axis acceleration would be stored as three 16-bit integers and an 8-bit scale value, which would take up a total of seven bytes compared to the eight which the timestamp occupies.

An alternative approach could be arranging all data points from a single source, i.e. a sensor, into a time series, where the first item in the series would hold the timestamp for the starting time and the interval that the measurements are sampled at. This would eliminate the overhead per data point and make it a constant size piece of metadata at the start of the series. The length of each series would then determine how many times this fixed size metadata overhead would be suffered. Benchmarking data compression algorithms and libraries and their suitability for this use case were left outside the scope of this thesis.

To decide on an approach, memory requirements were calculated for both options. The timestamps attached in either case were defined to be 64-bit integer values as this is the largest time value that can be fetched through Zephyr's timing interfaces. The system doesn't have a separate battery-powered real-time clock or other reliable timekeeping mechanisms which means that the timestamps are relative to the last system reboot. The timestamp indicates the milliseconds since the last system restart, which with a 64-bit integer's range would be millions of

years. The smaller up-time metric from Zephyr is represented as a 32-bit unsigned integer. This would result in a maximum representable up-time of almost 50 days, which is more reasonable for a wearable device and far exceeds the battery life objectives of the prototype. The calculation results for the storage formats are presented in Table 5 and Table 6. Sensors included in the calculations are the accelerometer, heart rate sensor, two temperature and humidity sensors, and an ADC with 8 measured channels. The default accelerometer sampling frequency was set at 25 Hz. A lower rate of 10 Hz was determined to be an adequate minimum if memory space was limited. All ADC channels are sampled at 0.1 Hz as are the temperature and humidity sensors. Heart rate is measured every second. In Table 6, the time series header size is 16 bytes, including the start time, sampling interval, and sample count of the series. The series length was set to one minute to avoid losing a large amount of data in the case of power loss.

Table 5. Memory requirements with different sensor combinations when storing each sample with its own individual timestamp.

Case sensors	Required Memory in 24h [MB]	Required Memory in 48h [MB]
All sensors	33.10	66.20
All sensors, Accelerometer 10 Hz	14.57	29.14
Heart Rate, Temperature&Humidity, Accelerometer 10Hz	13.65	27.30
Heart Rate, Temperature&Humidity, ADC	2.21	4.42
Heart Rate (only BPM), Temperature&Humidity	1.04	2.08

Table 6. Memory requirements with different sensor combinations when storing samples as a time series.

Case sensors	Required Memory in 24h [MB]	Required Memory in 48h [MB]
All sensors	15.40	30.80
All sensors, Accelerometer 10 Hz	6.75	13.5
Heart Rate, Temperature&Humidity, Accelerometer 10Hz	6.33	12.66
Heart Rate, Temperature&Humidity, ADC	0.955	1.91
Heart Rate (only BPM), Temperature&Humidity	0.29	0.58

As can be seen from Table 5 and Table 6, neither format is sufficiently compact to store two full days of all envisioned sensors. A large amount of acceleration data quickly fills the available 8 MB space. The time-series approach is favorable to the simple one since it can easily fit both the heart rate and temperature-humidity data as well as possible ADC and even lower sampling rate acceleration for shorter periods of time.

The QSPI flash memory needs to be written to and read from by the software. Zephyr provides multiple interfaces to storage devices. These interfaces include raw access to flash memory addresses as well as higher-level interfaces such as the file system and non-volatile storage APIs. Raw flash access provides the most control over the memory while burdening the software developer with the complexity of managing the addresses and block erasure manually. The non-volatile storage subsystem (NVS) stores data in id-value pairs and treats the flash as a circular buffer. Each id-value pair incurs eight bytes of metadata overhead (Zephyr Project, 2020a). While this interface is easy to use, it is not well-suited for storing time-series data, since each write to the id creates a new entry in the history of that id instead of modifying it in-place. It is also not possible to delete only a part of the entry, which is a necessary feature when data could be sent in batches off the device whenever a connection is available. The NVS may be useful for storing settings and configuration values in the future.

The file system interface in Zephyr allows accessing the flash memory much like a desktop system would. The interface contains functions for opening, closing, reading, writing, and deleting as well as moving to different points inside the file. The application developer can use a number of existing implementations of a file system or implement their own to match the interface. One of the existing implementations integrated into Zephyr is LittleFS, a small footprint fail-safe file system designed for microcontrollers (littlefs-project, 2020). It is supported on the nRF52840 QSPI flash memories and features protection against data corruption caused by a power loss and dynamic wear leveling to prolong the lifetime of the memory. These features along with the easy-to-use Zephyr interfaces make LittleFS a good fit for this prototype. The LittleFS files have some non-negligible storage overhead, which is tied to the file size. As the file size increases, the overhead decreases significantly (littlefs-project, 2019). It was decided that the overhead is acceptable considering the speed and convenience that using the file system provides. Furthermore, by keeping the time series for each measured metric in a single file dedicated to that metric, the overhead can be kept to a minimum since files will be large rather than small.

The flash memory space of 8 megabytes is partitioned in such a way that the LittleFS file system is assigned all but the last 4-kilobyte sector. The last sector is reserved for future use for metadata, configuration, and logging. Each measured metric is stored in its own file. The file system is initialized right after booting the device. The files are created at file system initialization time if they don't already exist. Each file is filled with time series headers and the associated data samples. At initialization, the program will seek the point in the file where the last entry was written and keep that index ready in Random-Access Memory (RAM) for the next read or write operation. Traversing the file at initialization also serves as a validity check for the time series headers. If the header information doesn't match the lengths of the series and the size of the whole file, an error will be revealed.

The samples provided by the sensor callback functions are buffered to wait for writing to the flash memory. Each metric has its own buffers in and out of flash and associated counter variables. The buffers are large enough to hold one time series, i.e. one minute of samples at each sensor's respective sampling rate. The buffer for writing, the *save buffer*, contains the samples with their timestamps attached. When the save buffer reaches its limit, a write is scheduled to the system workqueue. In the write operation, the file for the metric to be saved is first opened. The file cursor is then moved to the end and the end offset saved. Then, the time series header is

written to signal the start of a new series. The starting timestamp is the timestamp of the first sample of the save buffer. After the header, each save buffer sample's data fields are written one after the other to the file. In the end, the last written header information and offset are saved to variables in RAM. The closing of the file ends the write procedure and ultimately synchronizes the changes to the flash device. Once the write is completed, the save buffer starts filling again from index zero.

4.5 Transmitting Stored Data to the Gateway

Bluetooth Low Energy is the wireless transport used to transfer measured data to the Gateway mobile application. The Bluetooth protocol software implementation in Zephyr features, among others, interfaces for Bluetooth advertising, connections, and handling GATT services. Many of these interfaces accept callback functions as arguments, giving the application developer the possibility to choose how to react to events such as connection attempts or reading and writing values. In the prototype application, these interfaces are used privately and the application is provided with a simpler BLE interface with functions to:

- Initialize BLE and start advertising.
- Send bytes of data through GATT notification.
- Query the maximum data length, which can be sent in a GATT notification in the current connection.

The Bluetooth radio is initialized at startup. The device begins advertising its GATT services and is open to connections. Internally, the BLE module defines callbacks such that information can be obtained about the incoming connections and connection parameters. On connection, the device sends a request to the connected mobile device to exchange the MTU. In order to improve throughput, the data length extension feature needs to be enabled. The current Zephyr BLE settings allow a maximum MTU of 247 bytes and a LL data length of 251. The 2M PHY is also supported on the nRF52840 and it is switched to if the connecting device supports it.

When a connection is established, the two devices also exchange connection parameters, initially set by the Central device. The BLE Peripheral, the DAQ, can request a change in the parameters. However, the Central does not have to accept, although, in the case of the Gateway, it should be made to accept the parameters, if they are within the mobile implementation's limits to set.

The connection interval affects both throughput and power consumption and should be optimized with more testing in the future. The overall goal is to send as many packets in an interval and not get retransmissions on the next interval and thus throttle the throughput. The parameter exchanges are presented in Figure 7.

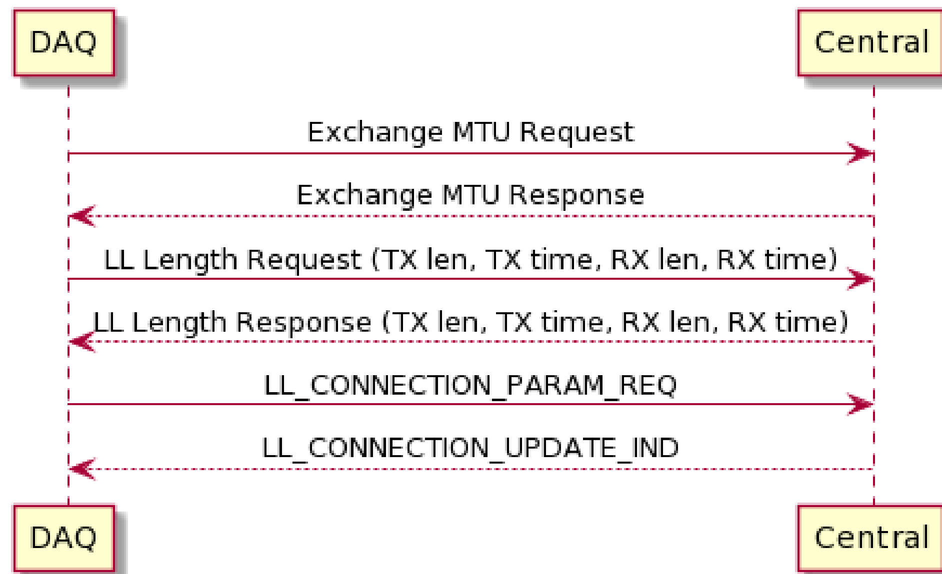

Figure 7. MTU exchange, Data Length Update, and connection parameter update initiated by the DAQ in the BLE Peripheral role. The BLE Central is a mobile device.

As can be seen from Figure 7, the DAQ takes the GATT Client role in order to exchange the MTU as soon as possible. The Central device responds. The DAQ then initiates the Data Length Update and receives the Central parameters in return. Finally, the DAQ may make a connection parameter change request when desired.

Once the connection is established and its parameters settled, the BLE module's function is centered around observing the CCCD in order to start sending notifications, if the Central device enables them. The GATT service used to send the measurement data is based on the Nordic UART Service. The Nordic UART Service is a custom GATT service that receives and writes variable-length byte data using one characteristic for receiving and one for transmitting. The GATT service supports writing from the mobile application side to the receiving characteristic, although this is currently not used for commands or configuration. The received data can be accessed from a *received* callback. Currently, the callback sends a timestamp and two small numeric values back to the mobile side for testing purposes. The NUS has been extended to include a callback, which is called when notifications are enabled or disabled. This change event

is used to trigger sending of data to the connected Gateway application. GATT notifications were chosen as the preferred operation for their better throughput potential since indications need to be acknowledged. Application-level acknowledgment of the reception of measurement data could be used to ensure the reliability of the transfer but as notifications are acknowledged at the Controller, it was decided to favor throughput. The NUS single-service single-characteristic approach was chosen for flexibility and simplicity of the scanning Central at this stage of development.

If notifications are enabled, the callback calls into the sensor application-level modules to stop sampling work. It will then send a firmware version string as well as the current uptime of the DAQ as notifications. These will be used by the Gateway application to know how to decode messages and to convert the device uptime to operating system time. For example, if device uptime is 120 000 ms at the time of enabling notifications and incoming data points have timestamps of 60 000 ms, the Gateway can deduce that the data was sampled 1 minute before the enabling time. After the time message, the callback will schedule send work to a dedicated workqueue in order to not block other operations using the system workqueue. On the opposite case, when notifications have been disabled, the callback schedules sampling work to start again.

The operating mode of the DAQ changes from sampling and saving data to loading old data and sending it over BLE when BLE notifications are enabled on the GATT service's transmitting characteristic. The send work item handler operates like a state machine and goes through a sequence of states as shown in Figure 8.

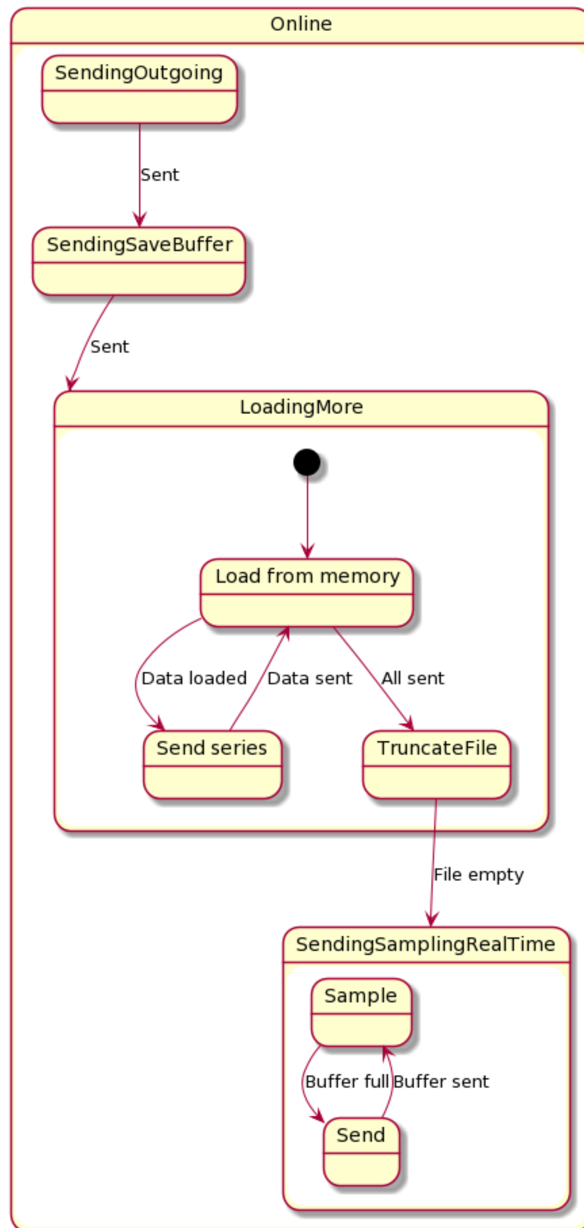

Figure 8. Data transmission state diagram. The system sends the most recent save buffer contents first and then loads data from the flash memory. After all of the data has been sent, the system returns to sampling.

Initially, when the BLE module detects a change in the notification status, it schedules sending work. The system is transitioned into the *SendingOutgoing* state. *Online* in Figure 8 depicts a pseudo-state, meaning that the notifications are enabled for the duration of its shown substates. However, the first state entered is actually *SendingOutgoing* and when notifications are disabled, a transition is made back to *Offline*.

In the *SendingOutgoing* state, individual samples from each metric's static *outgoing buffer* are sent as GATT notifications. The sending of measurement data is done by encoding it to JSON. JSON was chosen as the message format because it is human-readable and also used by the cloud interface. The messages consist of four JSON attributes: the metric name, starting timestamp in milliseconds, and the sampling interval in microseconds followed by one or more data points in the *values* attribute. The message construction is done by using the standard library function *snprintf* to insert values into a JSON string template. The size of the resulting JSON message is determined by the width of each numeric value in characters, not its storage space in bytes, e.g. the number 255 takes up three characters although just one byte would be enough to represent that value. An example JSON time series chunk for heart rate (HR) is shown below:

```
{
"metric": "HR",
"start_ts": 123456,
"interval_us": 1000000,
"values": [72, 73, 96, 68]
}
```

In the JSON message above, the starting timestamp is contained in the field *start_ts*. The interval is set to 1,000,000 microseconds, so one second. The BPM values form the *values* array.

To send status updates required in R-K3, *send_state_json* can be used with numerical status codes and a 30-character text field. The JSON metric for status updates is called *STAT*. The status messages omit the interval attribute and only include the timestamp.

When all outgoing samples have been sent, the system proceeds to send the samples in the save buffer. In the *SendingSaveBuffer* state, the newest samples are sent with their attached timestamps, as opposed to the previous state, which computes the timestamps from the starting timestamp and offset in the time series.

In *LoadingMore*, first, the maximum payload for notifications is obtained by calling the *bt_get_max_send_data_len* function of the BLE module interface. Then, the time series are loaded from QSPI flash memory and sent as notifications. The payload size is used to calculate how many values can fit into one JSON message. Each series is sent in maximum size chunks and this is repeated for each metric until all data from the flash memory has been sent.

The buffer for reading from flash, the *outgoing buffer*, holds the time series values, and the header information is stored and loaded to a separate variable. The timestamp for each member in the series is restored with the starting timestamp and the interval between samples from the header. Loading from the QSPI flash is performed as follows. First, the *load_series* function for a given metric is called. This function calls a file system helper function with the metric's *to_send_offset*. This offset is equal to the last written offset when starting to send values, as shown in Figure 9. The helper function seeks the file cursor to this offset from the beginning of the file and reads the header and the following data. It reads the data to the *outgoing buffer*, saves the header information to a static variable, and updates an outgoing sample counter variable with the read series length. Then, if the just loaded time series was the last series or there was nothing to read, the function raises the *sample_more* flag to signal moving on to the next metric, as illustrated in Figure 12. After each series has been sent, the *to_send_offset* for the metric is rolled back to the next series header, as presented in Figure 10. If an error occurs while sending a series, the file is truncated to the offset that is at the end of the unsent series, so that space is saved, but no data is lost. In LittleFS, files can be deleted only by truncating from the end, so in order to get the load offset, walking through the file from the beginning is needed. The QSPI Flash file system measurement file layout and operations are shown in sequence in Figures 9 - 12.

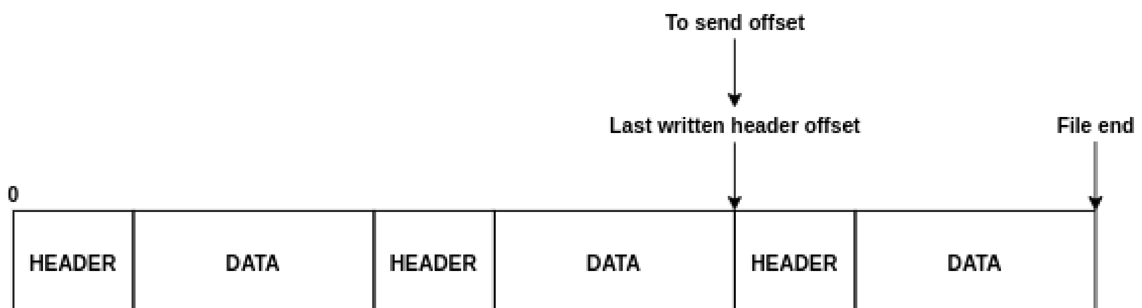

Figure 9. Flash memory data file layout after three stored time series.

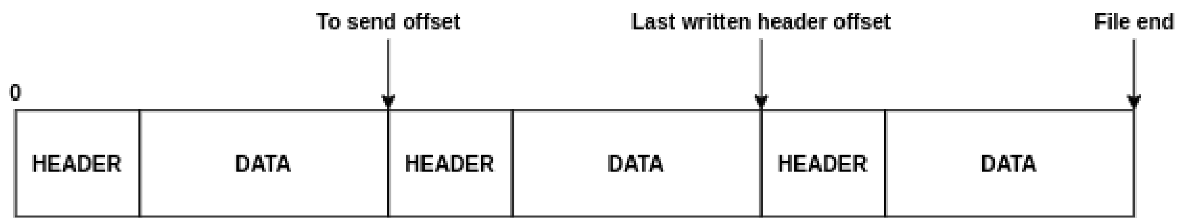

Figure 10. Flash memory offsets after one time series has been sent. The *to_send* offset has been moved back by one series.

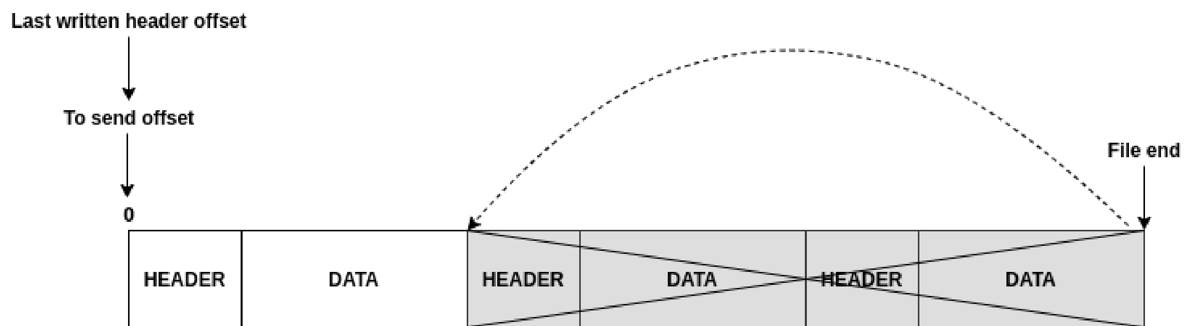

Figure 11. Flash memory offsets after two series have been sent. The last series hasn't been completely transmitted and the file has been truncated back down to this point.

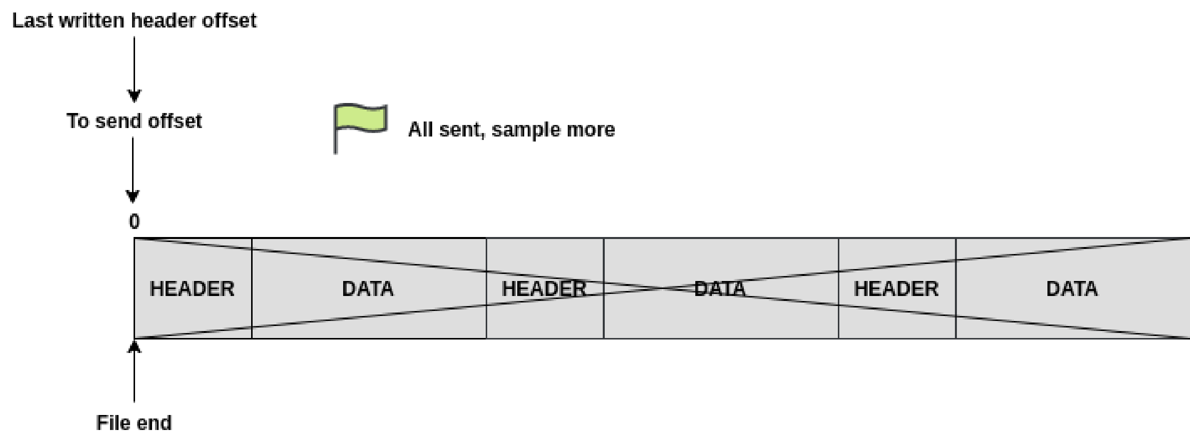

Figure 12. Flash memory data file offsets after all measurement data from the file has been sent. The offsets are zeroed and the file truncated to size zero. A flag variable is set to signal the end of this metric's data uploading.

The cause for an incomplete time series transmission and the truncation that follows, as illustrated in Figure 11, could be a common event such as loss of the Bluetooth connection or a rare occurrence such as the lack of free Bluetooth transmit buffers or a message formatting failure.

In both cases, the file is truncated only down to the end of the incomplete series, and the system transitions to the *Offline* state to log and handle errors if possible.

After all the metrics have been sent, the files are truncated to zero, and sampling work is started again. The system moves to the *SendingSamplingRealTime* state, where the buffering function checks if this state is active and schedules sending work instead of saving. This way a new series of samples is sent every minute as long as the notifications stay enabled. In this state, if sending should fail, the system will fall back to scheduling saving work for the buffer instead. Thus, no data is lost.

In this chapter, the design decisions of the prototype were presented. First, the choice of development tooling was discussed and then an overview of the system architecture was given. Finally, the implementation of each of the system components: sensing, storing of data, and transmitting data was discussed in detail. In the next chapter, the created prototype is analyzed and possibilities for future improvement are discussed.

5 ANALYSIS AND DISCUSSION

In this chapter, the implementation of the prototype is analyzed against the objectives and requirements of the project as laid out in chapters 1 and 2. Then, areas of future improvement are discussed.

5.1 Capabilities of the Prototype

The prototype is a (60mm x 30mm) mobile device with a Li-ion battery. It includes a Bluetooth module, an optical heart rate sensor, two temperature-humidity sensors, and an accelerometer.

The prototype currently implements the following features:

- Measures heart rate at 1 Hz.
- Measures 2 x temperature and humidity at 0.1 Hz.
- Saves measurement data persistently to QSPI flash memory.
- Sends measurement data to a connected BLE Central device.
- Sends status updates to a connected BLE Central device, when the system state changes.

The accelerometer on the prototype board is currently not supported in software as its high data rate proved problematic in terms of the available flash memory. The analog-to-digital converter was not integrated in time to the prototype hardware and was left out of the final software. A software driver implementation exists from an earlier prototype board, which may be used to develop a Zephyr-based driver in the future.

5.2 Review of the Prototype Requirements

A review of the fulfillment of the requirements introduced in chapter 2 is presented here.

5.2.1 Review of the Functional Requirements

First, the functional requirements presented in Table 1 are reviewed.

R-K1 DAQ measures biosignals and environmental variables with high quality

Out of the high-priority sensors, the heart rate and temperature-humidity sensor integration was completed during this thesis project. The accelerometer and analog-to-digital converter software drivers are at an experimental stage with some features such as accelerometer internal buffer processing missing. As previously mentioned, the accelerometer data rate also proved to be problematic with the current hardware. The heart rate sensor of the prototype contains its

own acceleration measurement that could be used in the meantime at a lower sampling rate. The largest contributing factor to the ADC integration was the time pressure. The driver integration is not estimated to consume significant effort going forward since Zephyr provides a good basis for the I2C and SPI bus communication that the sensor drivers need.

R-K2 DAQ sends measurements to the connected gateway application reliably
The prototype is able to transmit data to a connected mobile device through a Bluetooth Low Energy connection. The data is sent as GATT characteristic value notifications that contain a JSON-formatted message payload. Although notifications are not acknowledged at the application level, they are transmitted reliably at the radio physical layer. In the initial tests, no gaps in the received time-series data were observed. The implementation of the storage and transmission of the data further ensures that no data is deleted before it has been transmitted. Notifications provide improved throughput and thus shorten the time taken to offload the measurements from the prototype device to the BLE Central device.

R-K3 DAQ shall send the gateway a notification of system status changes
The system is able to notify the mobile application of state changes and errors through GATT notification. The status messages can be sent when there is an active BLE connection and the notifications are enabled. This allows the developer to mark parts of the software with status transmission function calls and provide status information wirelessly.

R-K4 DAQ shall change measurement parameters on command from the gateway
Currently, the system does not allow runtime parameter configuration. The configuration feature did not fit the schedule of the project and was left out of the scope as it was categorized as a convenience feature. Two-way communication between the Gateway application and the DAQ could be made possible through the receiving characteristic of the GATT service and its *received* callback. This callback could parse user commands and execute them when the system is in a suitable state.

R-K5 DAQ shall send the gateway a notification when the battery level drops below 10 %
The battery level notification did not fit the schedule of the project and was also left out of the scope as it was categorized as a convenience feature. Once the Li-ion battery has been tightly integrated into the prototype board, further development can start. The battery level may be added to the status messages, as its own metric or in a standard Battery Service.

R-K6 DAQ shall save measurement data to its non-volatile memory

The measurement data is stored persistently. Data is saved in minute-long batches, which ensures that large amounts of data are not lost, should an error occur. The file system library, LittleFS, provides features that protect against data corruption and prolong the lifetime of the memory. The flash memory has sufficient space to store heart rate, temperature-humidity, and future ADC measurement data. Measuring acceleration produces a large amount of data, whose storage format may need to be re-evaluated in the future.

R-K7 DAQ measured data points are traceable to a timestamp

Each measured sample includes a timestamp by the sensor driver. When the samples are stored into flash memory, they are stored as a time series with a start time and the sampling interval. The timestamps are restored based on the start time and interval when the measurements are loaded from flash memory. The method assumes a constant sampling rate, which is the case at this stage of the prototype. For the sampling frequencies of the current metrics, this can be deemed sufficient. Testing has shown that the samples preserve their time information through storage and transmission to the Gateway device. The Gateway device can transform the relative timestamps to date and time format with the time synchronization at the start of the transmission.

R-K8 DAQ shall be able to recover or try to recover from errors

The system should try to handle and recover from errors. The software modules in the prototype include basic error reporting. Most interfaces return an integer status code to indicate whether an error has occurred. A logging system with different log severity levels was used to report errors during development. The most severe logs are reported even in the smaller form factor hardware to help with solving possible problems. Fatal errors cause a system reboot since there is no graceful way to try to recover from them. Less serious or common error cases transition the system to a known offline state. There are improvements to be made in unifying error handling and error codes and better defining how the system should react to different conditions.

5.2.2 Review of the Non-Functional Requirements

The qualities of the implemented prototype are analyzed as follows:

Portability and Learnability

A comprehensive documentation package was created for the software prototype. The documentation includes a user manual, development software setup instructions, a software

architecture document, and source code commenting. The development software setup instructions were tested to work on both Windows and Linux operating system environments. Using the instructions and the user manual, it is possible for new users to collect the stored data through a mobile application and develop new software features.

Comfortability and Reliability

Due to project timing constraints and a lack of dedicated testing resources, the prototype hardware-related tests could not be carried out. The comfortability and reliability cannot be commented on as the hardware was not made into a strapped, wearable device on time.

Battery life

The battery life was tested with a coin cell battery, which had four times the envisioned capacity of the Li-ion battery and different characteristics. The battery life in the initial tests was slightly over 96 hours, which adjusted to a quarter of the capacity would correspond to a 24-hour operating span. Optimizations to the power consumption were not yet made in software because of the time constraints and a lack of testing resources with the majority of focus on the core feature development.

In summary, the prototype fulfills its core responsibilities of measuring, storing, and sending data to a good extent. Compromises regarding the number of sensors and lower priority features had to be made due to time constraints, but the prototype serves as a good basis for future development. There is a need for further testing and measurements that will guide the optimization of power consumption and reliability.

5.3 Future Improvement Areas

A first prototype cannot include every product feature. Some features were left outside the scope of this study from the beginning and some objectives had to be adjusted to fit in the thesis schedule. The following improvement areas were identified during the project:

- Testing
- Addition of missing ADC and acceleration measurement
- Storage optimizations
- Power consumption measurement and optimization
- Bluetooth throughput measurement and optimization
- Missing convenience features

- Firmware updates
- Architectural improvements

Since the project was lacking dedicated testing resources, the current prototype should first be tested by another person to catch mistakes that may have been overlooked. In the future, the testing framework capabilities of Zephyr could be researched to automate aspects of hardware and software testing.

The focus of new development after the current prototype has been tested should be the missing sensors, the ADC and accelerometer, which were left out due to schedule and storage size issues. The driver development for these sensors has a good basis in Zephyr and the earlier prototype boards. The addition of the accelerometer necessitates either more frequent offloading of data to the gateway or storage optimizations. The storage format could be researched and experimented with further. The suitability and performance of data compression libraries to this application could be studied in detail. It might also be that a bigger storage device is needed to include all metrics.

Power consumption is another point of interest in future development. Firstly, the system should be measured and profiled. The optimization could then concentrate on the consumption hot spots. Some parts to look at could be the radio usage: transmit power and advertising interval, though they should be adjusted taking into account the range and responsiveness requirements. Lower processor sleep modes could be utilized and peripherals turned off if a wake-up signal was used to bring the system back to sampling. An accelerometer activity interrupt might be suitable. The time delay between signal and full sampling readiness would need to be tested and evaluated when putting the processor to sleep.

Another aspect worth measuring could be the Bluetooth throughput. Bluetooth transmission of data messages should be reliable but preferably also fast. The duration of the data offloading affects both the power consumption and the user experience if the offloading is done in the "foreground" and the user wants to observe results immediately. Both the Bluetooth connection parameters and the message serialization format should be experimented with. Could another way to format JSON or perhaps another format entirely bring advantages?

The features regarded as nice-to-have, such as the battery level notification or runtime configuration, are worth keeping in the development backlog until it is felt that higher priority tasks have been completed or a convenience feature is needed to speed up testing or development. Wireless firmware updates may be an example of a convenience or even an essential feature, once devices are deployed to field testing. On the other hand, firmware updates bring up security considerations and the security aspect of the system should be polished in any case at the latest before deploying devices past the laboratory environment.

Finally, the structure of the application and the programming practices can always be improved upon. Keeping the software extendable requires attention and maintenance. Programming errors can be incrementally fixed, guided, and backed up by testing. Attempting to keep software modules loosely coupled with well-defined interfaces makes it easier to change module implementations and develop new features.

5.4 Prototype Design and Implementation Key Takeaways

Below are a few insights which were found during this project that may be useful for similar development projects in the future.

First, it was deemed helpful to think about the system in a data-first manner. When the sources and production rates of data, the operations on the data, and the points of consumption of the data in the system were identified, it simplified the software architecture design process. Furthermore, data transformations and formats can be tested in isolation in an early phase of the project if the data sources and sinks are simulated. This allows verifying key assumptions as fast as possible.

Second, it is good to take low-power operation into consideration when choosing the sensors and peripherals of the hardware. Preferably, all sensors would include a data interrupt, a low-power state, and a wake-up mechanism. This would give the software developer more optimization room.

Finally, the Zephyr RTOS and its ecosystem of software libraries and tools were tested to be a good basis for rapid development. It will be interesting to observe the project's growth and it may be worthwhile to check the Zephyr support for one's intended microcontroller family if the application warrants the use of an real-time operating system.

6 CONCLUSIONS

In this thesis, a wearable prototype of a data acquisition device was designed and implemented. The technical requirements for the prototype were gathered with the client and the limitations of the scope of the prototype were also discussed.

The available software development tools were evaluated and the Nordic Connect SDK was chosen for the implementation. The implementation was guided by the core functions of the data acquisition device: measuring quantities with its sensors, storing the measured data on the device, and transmitting the stored data to a connected gateway device for further processing.

The prototype measures heart rate, temperature and relative humidity, and stores that data into non-volatile memory. When connected to a gateway device, the stored data and any status messages are sent to the connected device. The prototype software also implements reliable timestamping of samples and basic error handling and logging.

The resulting prototype was documented and analyzed. The software handles its core functional requirements well according to the requirements and is thoroughly documented for simple setup and further development. Compromises on the number of sensors and convenience features had to be made due to the thesis scope and shortage of storage space. There is room for optimization in power consumption and wireless data throughput as well as the storage hardware and software solution in the future.

The key takeaways for similar projects are to focus on the flow of data in the software design, verify assumptions about the application early, and choose sensors and peripherals that are suited for low-power operation.

REFERENCES

Aimonen, P. (2021). *Nanopb - protocol buffers with small code size*. Available: https://jpa.kapsi.fi/nanopb/.

Beningo, J. (2017). *Reusable Firmware Development: A Practical Approach to APIs, HALs and Drivers*, 1st edn. Berkeley, CA: Apress. ISBN 978-1-4842-3297-2.

Bluetooth SIG (2021). *Bluetooth Core Specification Version 5.3*. Bluetooth SIG. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=521059.

Bormann, C. (2020). *CBOR — Concise Binary Object Representation*. Available: https://cbor.io/.

Bulić, P., Kojek, G., & Biasizzo, A. (2019). Data Transmission Efficiency in Bluetooth Low Energy Versions. *Sensors*, 19(17), p. 3746. doi:10.3390/s19173746, Available: https://www.mdpi.com/1424-8220/19/17/3746. Publisher: Multidisciplinary Digital Publishing Institute.

Ecma International (2017). *ECMA-404: The JSON data interchange syntax, 2nd edition*. Ecma International. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

Furuhashi, S. (2019). *MessagePack: It's like JSON. but fast and small.* Available: https://msgpack.org/.

Gomez, C., Oller, J., & Paradells, J. (2012). Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. *Sensors*, 12(9), pp. 11734–11753. doi:10.3390/s120911734, Available: https://www.mdpi.com/1424-8220/12/9/11734. Publisher: Molecular Diversity Preservation International.

Google Inc. (2021). *Protocol Buffers*. Available: https://developers.google.com/protocol-buffers.

Hamerski, J.C., Domingues, A.R., Moraes, F.G., & Amory, A. (2018). Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs. In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 773–776.

Hamida, S.T.B., Hamida, E.B., & Ahmed, B. (2015). A New mHealth Communication Framework for Use in Wearable WBANs and Mobile Technologies. *Sensors*, 15(2), pp. 3379–3408. doi:10.3390/s150203379, Available: https://www.mdpi.com/1424-8220/15/2/3379. Publisher: Multidisciplinary Digital Publishing Institute.

John Dian, F., Vahidnia, R., & Rahmati, A. (2020). Wearables and the Internet of Things (IoT), Applications, Opportunities, and Challenges: A Survey. *IEEE Access*, 8, pp. 69200–69211. ISSN 2169-3536, doi:10.1109/ACCESS.2020.2986329.

Kvaale, B. (2020). *nRF Connect SDK: Next generation SDK for Nordic wireless solutions*. Nordic Semiconductor. Available: https://devzone.nordicsemi.com/cfs-file/__key/communityserver-discussions-components-files/4/nRF_5F00_Connect_5F00_SDK_5F00_webinar.pdf.

littlefs-project (2019). *littlefs DESIGN.md*. Available: https://github.com/littlefs-project/littlefs/blob/v2.2.0/DESIGN.md.

littlefs-project (2020). *littlefs README.md*. Available: https://github.com/littlefs-project/littlefs/blob/v2.2.0/README.md.

Macronix International Co., Ltd. (2020). *MX25R6435F: Ultra Low Power, 64M-Bit [x 1/x 2/x 4] CMOS MXSMIO (Serial Multi I/O) Flash Memory*. Macronix International Co., Ltd. Available: https://www.macronix.com/Lists/Datasheet/Attachments/7913/MX25R6435F,%20Wide%20Range,%2064Mb,%20v1.5.pdf. Rev. 1.5.

Nakamura, Y., et al. (2017). SenStick: Comprehensive Sensing Platform with an Ultra Tiny All-In-One Sensor Board for IoT Research. *Journal of Sensors*, 2017, p. e6308302. ISSN 1687-725X, doi:10.1155/2017/6308302, Available: https://www.hindawi.com/journals/js/2017/6308302/. Publisher: Hindawi.

Nordic Semiconductor (2018). *nRF5 SDK v15.0.0 Release Notes*. Nordic Semiconductor. Available: https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v15.0.0/index.html.

Nordic Semiconductor (2021a). *About the nRF Connect SDK — nRF Connect SDK 1.6.0 documentation*. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/1.6.0/nrf/introduction.html.

Nordic Semiconductor (2021b). *Bluetooth Low Energy | Development Software: Great hardware deserves great software*. Available: https://www.nordicsemi.com/Products/Bluetooth-Low-Energy/Development-software.

Nordic Semiconductor (2021c). *nRF52840 Product Specification v1.2*. Nordic Semiconductor. Available: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.2.pdf.

Nordic Semiconductor (2021d). *Releases · nrfconnect/sdk-nrf*. Available: https://github.com/nrfconnect/sdk-nrf/releases.

Petersen, B., Bindner, H., You, S., & Poulsen, B. (2017). Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the Internet of Things. In: *2017 Computing Conference*, pp. 1339–1346.

Statista (2020). *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030, by use case*. Available: http://www.statista.com/statistics/1194701/iot-connected-devices-use-case/.

Townsend, K., Cufí, C., Akiba, & Davidson, R. (2014). *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*, 1st edn. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc. ISBN 978-1-4919-4951-1.

Zephyr Project (2020a). *Non-Volatile Storage (NVS)*. Available: https://docs.zephyrproject.org/2.4.0/reference/storage/nvs/nvs.html.

Zephyr Project (2020b). *Workqueue Threads*. Available: https://docs.zephyrproject.org/2.4.0/reference/kernel/threads/workqueue.html.

Zephyr Project (2020c). *Zephyr Project | Benefits*. Available: https://zephyrproject.org/benefits/.