Optimizing Smart Agriculture for Chicken Egg Hatch Environments Using the Internal Sensors of the Nordic Semiconductor Thingy:53

Nofri Alhadi - 23188901

Department of Computer Science and Information Systems

Faculty of Science and Engineering

University of Limerick

Supervised by:

Prof. Dr. Tiziana Margaria

Amandeep Singh

Submitted to the University of Limerick

in partial fulfilment of the requirements for the degree of

Master of Science in Artificial Intelligence and Machine Learning

August 2024

Declaration

I hereby declare that I have written this thesis without the prohibited assistance of third parties and without utilizing any aids other than those specified. Any ideas or information obtained directly or indirectly from external sources have been duly acknowledged. This thesis has not been previously submitted in an identical or similar form to any Irish or foreign examination board.

The thesis work was carried out in 2024 under the supervision of Prof. Dr. Tiziana Margaria and Amandeep Singh at the University of Limerick.

Nofri Alhadi
Limerick, 2024

Acknowledgements

I would like to express my deepest gratitude to everyone who has supported me throughout this project. First and foremost, I would like to thank my supervisors, Prof. Dr. Tiziana Margaria and Amandeep Singh, for their invaluable guidance, encouragement, and expertise, which have been instrumental in shaping this work. Your patience and insightful feedback have genuinely enriched this study.

I would also like to extend my heartfelt thanks to my wife, Rizki Amalia, for her unwavering support, understanding, and love throughout this journey. Her encouragement and belief in me have been a constant source of strength, especially during the most challenging moments of this project.

Additionally, I would like to acknowledge my colleagues and peers in the Master of Science in Artificial Intelligence and Machine Learning 2023 for their constructive discussions and feedback.

Lastly, I want to recognize the Department of Computer Science and Information Systems, Faculty of Science and Engineering, University of Limerick, for providing the necessary resources and support that made this project possible.

Thank you all for your significant contributions to this work.

Dedication

To my wife, Rizki Amalia, whose endless love, support, and encouragement have been my constant strength and inspiration throughout this journey. And to my supervisors, Prof. Dr. Tiziana Margaria and Amandeep Singh, whose invaluable guidance, wisdom, and patience have been instrumental in completing this work. This thesis is dedicated to all of you with heartfelt gratitude.

Abstract

The successful incubation of chicken eggs requires precise control of environmental conditions throughout the entire process. This proposal for a thesis aims to investigate how the Nordic Semiconductor Thingy:53, an IoT device equipped with a range of internal sensors, can be utilized to optimize and automate the management of egg-hatch environments. The research will utilize Thingy:53's BME688 sensor to monitor temperature, humidity, pressure, and gas (air quality) and the BH1749 light sensor to detect light levels. The goal is to develop a reliable real-time monitoring and control system by reducing human intervention through automated data collection, real-time alerts, and comprehensive data analytics, ultimately enhancing hatch rates and operational efficiency. Machine learning algorithms will predict optimal hatching conditions based on historical data, enabling proactive adjustments to the incubation environment. Integrating with cloud platforms will allow long-term data storage and advanced analytics, providing valuable insights for ongoing improvement. This study will assess the effectiveness of Thingy:53 in maintaining optimal hatching conditions and its potential as a cost-effective, sustainable tool in poultry operations. The results will contribute to the advancement of smart agricultural practices and lay the groundwork for future innovations in the industry.

Contents

List of Tables

List of Figures

1

Introduction

The agricultural field is currently experiencing change due to technological progress, especially in the area of the Internet of Things (IoT). IoT technology allows for the gathering and assessing large volumes of data from different sensors, providing immediate insights and the ability to automate processes. In the poultry sector, one crucial use of IoT is to improve the egg-hatching environment for better hatch rates, reduced human involvement, and improved operational efficiency.

Creating the right conditions for hatching chicken eggs involves precise management of factors like temperature, humidity, light, and air quality. Conventional methods often rely on manual monitoring and adjustments, which can be labour-intensive and prone to errors. Implementing intelligent agricultural methods utilizing IoT devices presents a promising solution to these challenges.

1.1 Problem Statements and Objectives

Problem Statements

Despite the potential advantages of IoT in agriculture, thorough research is necessary to authenticate its efficiency in particular applications, such as egg incubation. Key concerns encompass:

- Ensuring precise regulation of environmental conditions to maximize the hatching success rate.

- Decreasing the necessity for manual involvement through the implementation of automation.
- Offering real-time monitoring and notifications to address any deviations promptly.
- Employing data analytics to acquire insights and continually enhance incubation.

This study seeks to overcome these challenges by investigating the utilization of the Nordic Semiconductor Thingy:53, an advanced IoT device equipped with various internal sensors, to streamline and automate the supervision of egg-hatching environments.

Objectives

The project aimed to develop an IoT-based system using the Nordic Semiconductor Thingy:53 to monitor and control an egg-hatching environment. It involved integrating sensors to gather environmental data, applying data analytics and machine learning to optimize conditions, and assessing the system's effectiveness in enhancing hatch rates and efficiency.

The objectives are:

1. Develop a prototype IoT-based system for real-time monitoring of the egg-hatching environment using the Thingy:53.
2. Incorporate various sensors of Thingy:53 (e.g., temperature, humidity, pressure, light, and air quality) to gather comprehensive environmental data.
3. Apply data analytics and machine learning methods to optimize hatching conditions.

1.2 Structure of the Thesis

The outline of the thesis will be structured as follows:

- **Chapter 1: Introduction**

  This chapter introduces smart agriculture, focusing on optimizing chicken egg-hatching environments using IoT technology. It outlines the research objectives, the study's significance, and the project's scope.

- **Chapter 2: Related Work**

  This chapter examines the current literature on IoT in agriculture, focusing on egg hatching and relevant technologies.

- **Chapter 3: Design Process and Timeline**

  This chapter outlines the systematic approach to designing the egg-hatching optimization system, covering the sections Requirement Analysis, System Design, and Prototyping.

- **Chapter 4: System Implementation**

  This chapter outlines the practical development of the system based on the design specifications, encompassing the implementation process, hardware and software setup, and integration of various system components.

- **Chapter 5: Data Analytic Workflows**

  The analytic workflows entail creating and assessing machine learning models to predict ideal conditions using sensor data from the Thingy:53 device.

- **Chapter 6: Results and Discussion**

  This chapter discusses the research findings, including data analysis and interpretation.

- **Chapter 7: Conclusion and Future Work**

  Summarizes the research outcomes, addresses limitations, and proposes future research directions.

1.3 Data Source and Generative AI

Data Source

The data source used in this study is the sensor data from Nordic Thingy:53 [1], which collects environmental data such as temperature and humidity using the onboard BME688 sensor. Generating a synthetic dataset for train and test with conditions "Ideal" and "Non-Ideal" egg-hatch environment to create data that mimics the conditions present during the setter and hatcher phases, considering temperature and humidity parameters [4].

Generative AI

The AI tool I used for this study is ChatGPT from OpenAI [5], which helped me understand code syntax and its usage. I also used Grammarly [6] for spelling and grammar checkers and to improve sentence clarity.

2

Related Work

This chapter examines the pertinent technologies, application areas, and tools that serve as the basis of our project. We start by investigating the fundamental technology that supports the design and operation of the system, followed by an overview of the specific domain of application where these technologies have the most significant impact. Next, we delve into the individual hardware and software elements, including the Nordic Thingy:53 and DIME (Device Information Model Editor), both of which hold essential roles in our system. Lastly, we explore the incorporation of Thingy:53 within the broader scope of IoT-based monitoring solutions, emphasizing its capabilities and importance in our project.

2.1 Adopted Technology

The Adopted Technology section overviews the key technological foundations used in our project. We'll start with a detailed look at the Development Environment, which includes the essential tools and platforms for building and deploying our system. After that, we'll discuss the role of IoT (Internet of Things) in enabling real-time monitoring and data collection from connected devices. Lastly, we'll delve into Artificial Intelligence (AI), which is crucial for analyzing sensor data and optimizing the conditions for the egg-hatching environment. Each of these technologies plays a critical role in ensuring the effectiveness and efficiency of our system.

2.1.1 Development Environment

A development environment consists of processes and tools that software developers utilize to write, test, and debug their code. This encompasses integrated development environments (IDEs), programming languages, libraries, and frameworks. Contemporary development environments offer a smooth interface for developers to construct robust applications, particularly in the IoT domain, where multiple components must interact efficiently.

The Thingy:53 by Nordic Semiconductor is a sophisticated prototyping platform for IoT applications. It has diverse sensors and connectivity options, making it well-suited for creating and testing IoT solutions. The Thingy:53 supports Bluetooth Low Energy (BLE) and operates on the nRF5340 SoC [1].

Visual Studio Code (VS Code) is a well-liked, open-source code editor that supports various programming languages and development tools. It provides a versatile environment for coding, debugging, and version control [7].

The nRF Connect SDK is an extensive suite offered by Nordic Semiconductor for creating applications on its hardware platforms. It encompasses libraries, sample code, and tools that simplify the development of IoT applications and support numerous connectivity protocols [8].

Zephyr OS is a scalable, real-time operating system optimized for resource-constrained devices. The nRF Connect SDK supports Zephyr OS, which offers a strong foundation for building secure and reliable IoT solutions [9].

DIME (DyWA Integrated Modeling Environment) is a tool used to develop web applications using a model-driven approach. It emphasizes agile development and rapid prototyping, utilizing graphical domain-specific languages to address various aspects of web applications, including data modelling, business logic, user interface, and access control [2].

MongoDB is a flexible NoSQL database designed for scalable data storage. It is particularly suited to handling the diverse and dynamic data generated by IoT devices, enabling efficient storage and retrieval of sensor data for real-time analysis and decision-making [10].

2.1.2 IoT

The Internet of Things (IoT) concept pertains to a system of physical items integrated with sensors, software, and other technologies to communicate and share data with other devices and systems via the Internet. IoT devices are instrumental in developing intelligent agriculture by delivering real-time data, automating tasks, and enhancing operational effectiveness [11].

The Internet of Things (IoT) concept involves linking regular objects to the Internet to communicate and interact independently. Its goal is to connect the physical and virtual realms through Machine-to-Machine (M2M) communications [12].

IoT Architecture

The Internet of Things (IoT) architecture is crucial for effectively implementing the concept. One prominent framework for IoT architecture is the Three-Layer Architecture, which encompasses the perception, network, and application layers. This structure ensures that data is efficiently captured by sensors, transmitted through the network, and utilized by applications for decision-making and action.

An expanded version of this architecture is The Five-Layer Architecture, which introduces additional layers for processing, transport, and business functions. These added layers enhance the capabilities of the IoT system, enabling more sophisticated data processing, streamlined data transportation, and improved business integration.

In IoT applications, sensors and actuators play a pivotal role in collecting data and enabling actions based on that data. These components are instrumental in gathering real-world information and facilitating appropriate responses within the IoT ecosystem.

Communication in IoT spans various interactions, such as device-to-device, device-to-cloud, and device-to-gateway. This multifaceted communication network ensures seamless data exchange and efficient coordination within IoT environments.

Communication Models in IoT

- **Device-to-Device Communications**

Devices directly communicate with each other using protocols like Bluetooth, Z-Wave, or ZigBee without relying on an intermediary server. This enables direct connections across various networks, including IP and the Internet.

- **Device-to-Cloud Communications**

  In this model, IoT devices connect directly to the Internet cloud through traditional Ethernet or WiFi connections. An application service facilitates data exchange and message control, establishing a pathway from the device to the cloud services.

- **Device-to-Gateway Communications**

  In this model, IoT devices communicate through an Application Layer Gateway (ALG) as an intermediary between the device and cloud services. The ALG offers security, data translation, and a channel for accessing cloud services [12].

IoT Device

IoT devices, such as the Nordic Semiconductor Thingy:53, have various sensors for monitoring environmental conditions and transmitting data for analysis. Specifically, the Thingy:53 has sensors like the BME688 for measuring temperature, humidity, pressure, and air quality and the BH1749 for detecting light levels. These sensors allow precise monitoring and regulation of the environment, which is crucial for applications like egg incubation [11].

Incorporating IoT devices in agriculture has resulted in the advancement of intelligent farming techniques. These devices support data-based decision-making, reduce the necessity for manual involvement, and enhance the overall efficiency and sustainability of farming operations. By leveraging IoT technology, farmers can optimize resource utilization, boost productivity, and ensure better management of livestock and crops.

2.1.3 Artificial Intelligence (AI)

Artificial intelligence encompasses various fields and seeks to emulate human intelligence in robots to facilitate learning and problem-solving. Research scientists

and experts are harnessing AI technology to combat productivity challenges in the agricultural industry. AI is pivotal in selecting appropriate crop varieties, managing pests and diseases, estimating yield, and predicting agricultural commodity prices. The agricultural sector leverages AI techniques such as deep learning, machine learning, and image processing to address its unique challenges effectively. AI solutions allow real-time monitoring of weather patterns, temperature fluctuations, water usage, and soil conditions, thereby enhancing decision-making processes [13].

Machine Learning Data Classification Model for Data Classification: Logistic Regression and Support Vector Machine (SVM) Model Evaluation Metric

AI and IoT Integration

Artificial Intelligence (AI) and the Internet of Things (IoT) are increasingly combined in various industries, driving the fourth industrial revolution. AI technologies such as Machine Learning and reasoning are anticipated to transform IoT implementations by facilitating enhanced decision-making through data analysis. Integrating AI and IoT presents obstacles such as dependability and real-time data analysis. Nevertheless, when merged, the substantial data produced by IoT and AI technologies becomes potent tools, with AI relying on big data to produce meaningful outcomes. The amalgamation of AI and IoT is called "cognitive IoT" [14].

Tzafestas (2018) states that AI (artificial intelligence) and IoT (Internet of Things) are closely connected technologies that complement one another. IoT gathers extensive data, which AI processes to uncover insights, make decisions, and facilitate automation. AI enriches IoT by supplying the intelligence necessary to analyze data, recognize patterns, and make well-informed decisions. Integrating AI and IoT is crucial for optimizing the potential of interconnected devices and systems across various applications [15].

2.2 Application Domain

The Application Domain section focuses on the specific context in which our system is applied. It begins by examining Poultry Egg Incubation and Hatchery, providing an overview of the processes involved in eggs' natural and artificial hatching. We then discuss the advantages and disadvantages of natural and artificial incubation

methods, comparing the benefits and challenges of each approach. Finally, we explore the factors for ensuring optimal hatchability during artificial incubation, identifying the key conditions that must be controlled to achieve the highest success rates in artificial incubation. This section sets the stage for understanding the practical importance of the technologies and methodologies applied in our project.

2.2.1 Poultry Egg Incubation and Hatchery

Poultry egg incubation refers to hatching eggs using a broody hen or an incubator machine. Natural incubation entails a broody hen sitting on the eggs, whereas artificial incubation uses specialized equipment. Hatchery management involves handling, storing, incubating, hatching, and rearing chicks. Large operations utilize multi-stage incubators, while single-stage incubators accommodate varying numbers of chicks hatched. Successful hatches and chick quality depend significantly on proper management [4].

2.2.2 Advantages and Disadvantages of Natural and Artificial Incubation Methods

When comparing natural and artificial incubation methods, it's essential to consider their advantages and disadvantages.

Natural incubation is a cost-effective method for hatching eggs. It harnesses birds' instincts to sit on and warm their eggs until they hatch. This method requires minimal equipment and can be done without electricity or technical expertise. However, natural incubation is time-consuming and demands a significant amount of effort. It also relies on the brooding bird to create and maintain the conditions for successful hatching.

On the other hand, artificial incubation offers a more efficient approach to hatching eggs. It allows for precise control over temperature, humidity, and egg turning, which can result in higher hatch rates. Artificial incubators can also hatch many eggs simultaneously, making them a practical choice for commercial hatcheries. However, the downside of artificial incubation is the initial investment

required to purchase the equipment. Quality egg incubators can be costly, and a learning curve is involved in operating and maintaining them effectively.

In summary, while natural incubation is cost-effective but time-consuming and labour-intensive, artificial incubation is efficient but entails a higher upfront financial investment.

2.2.3 Factors for Ensuring Optimal Hatchability During Artificial Incubation

- **Temperature**

  During the first 18 days in the setter phase, the recommended temperature ranges for setters and hatchers (incubators set temperature) are between 37.2 and 38.2°C. For the last three days in the hatcher phase, the temperature should be lowered to 36.0 and 36.5°C. The ideal temperature range for chicken embryo development is between 37.2 and 38.6°C.

  Maintaining a consistent temperature throughout the incubation period is crucial, with a slight decrease during the hatching phase. Deviating from the optimal temperature range can reduce hatchability, poor chick quality, and embryonic mortality. The recommended set temperatures for setters and hatchers may differ depending on the incubator manufacturer.

  The embryo's temperature is just as crucial as the incubator's set temperature. The optimal embryo temperature range is between 37.2 and 38.6°C. For the first ten days of the incubation period, the embryo temperatures should be closer to the lower end of the optimal range (37.2°C). For the remaining period (11-21 days), the embryos' temperature should be closer to the upper end (38.6°C) of the optimal range [4].

- **Relative Humidity**

  The air's moisture level and relative humidity (RH) are crucial for egg development and hatching. If the RH is too low, the egg's contents can dry out quickly, resulting in lower hatch rates and the production of smaller, lower-quality chicks. It is recommended to maintain an RH of `55-65%` during the

first 18 days and then increase it to around 70% after the eggs are transferred to the hatcher. As the chicks begin to pip and hatch, the RH should be raised to 75-80%. Maintaining the proper humidity levels is crucial for ensuring the right moisture for the embryo's development and successful hatching [4].

- **Ventilation**

  Proper air circulation is essential during artificial incubation to maintain a consistent oxygen supply and remove carbon dioxide and moisture. It is optimal for developing embryos to receive fresh air containing 21% oxygen, and a decrease in oxygen levels can reduce hatchability. Inadequate ventilation can result in heightened CO2 levels, harming developing embryos. It is advisable to adjust ventilation levels according to the age of the embryos, with lower rates during the first week and higher rates in the following weeks. During hatching, ventilation should be reduced to uphold high humidity levels for successful incubation [4].

- **Position of Eggs and Turning**

  The orientation of eggs in the trays and the method of turning them are critical for developing embryos and their ability to hatch. When eggs are placed with the larger end facing upwards, it ensures that over 90% of embryos have their heads positioned near the air cell. Placing them with the smaller end facing upwards can result in developmental issues. In commercial hatcheries, eggs are typically placed with the large end facing upwards. They are rotated along their long axes to prevent embryos from sticking to the shell membrane, which helps reduce mortality rates. To prevent embryos from adhering to the shell membrane, it is recommended to rotate the eggs every 1 to 2 hours during the initial 18 days of incubation [4].

2.3 Nordic Thingy:53

The Nordic Thingy:53 is a versatile multi-sensor prototyping platform for Matter, embedded machine learning, and various wireless IoT products. This platform is grounded on the nRF5340 System-on-Chip (SoC), our top-tier dual-core wireless

SoC. Armed with motion, sound, light, and environmental sensors, it is an ideal launchpad for constructing proof-of-concepts and swiftly cultivating new prototypes.

Figure 2.1: Nordic Thingy:53 [1]

In this latest release of the Nordic Thingies, the case of the Thingy:53 boasts an entirely fresh design founded on the familiar form factor from the Thingy lineup. While preserving the compact square shape and slender profile, a door has been introduced to the casing, providing convenient access to the on/off switch and the external connectors without removing the outer shell. Adding a Stemma/Qwiic/Groove-compatible external connector elevates the user experience when utilizing the Thingy:53 with or without external hardware accessories.

The application core of the nRF5340 SoC, featuring the Arm Cortex-M33 processor, assures Thingy:53's ability to manage intensive embedded machine learning tasks without impacting wireless connectivity. Clocking in at 128 MHz, the application core delivers optimal performance, accompanied by ample storage with 1 MB of flash storage and 512 KB RAM to accommodate diverse applications. The wireless connectivity is independently handled by another Arm Cortex-M33 core, operating at a reduced 64 MHz for energy-efficient operation without infringing on the computational resources of the application core.

It's debugging, and the current measurement board complements each Thingy:53 in the packaging. This compact PCB grants easy access to otherwise inaccessible pins, serving as a valuable troubleshooting accessory, especially when combined with Power Profiler Kit II or other standalone debugging or power-analyzing hardware [1].

2.3.1 Key Features

The key features of this battery-powered prototyping platform for Matter and machine learning on the nRF5340 SoC include support for multiple wireless standards such as Bluetooth LE, Bluetooth mesh, Thread, and Zigbee. It has environmental sensors for temperature, humidity, air quality, and pressure and a colour and light sensor. Additionally, it features a low-power accelerometer, a 6-axis inertial measurement unit (IMU), a buzzer, and a PDM microphone. The platform also offers a connector for additional external boards and accessories and is powered by a USB-C rechargeable Li-Po battery.

Two mobile apps, namely the nRF Edge Impulse mobile app for embedded machine learning and the nRF Programmer mobile app for quickly flashing firmware on the go, are also included. The platform is powered by a high-performance 128 MHz Arm Cortex-M33 application core and an ultra-low-power 64 MHz Arm Cortex-M33 network core. It houses a multi-protocol radio with Bluetooth LE, Bluetooth mesh, Thread, and Zigbee support and employs a highly efficient nPM1100 PMIC for better battery life. Additionally, it features a full power path for seamless switching between charging and battery operation and incorporates an nRF21540 FEM RF front end for extended range and increased link robustness.

2.3.2 Applications

The applications of this technology are diverse and widespread. One prominent application is machine learning, where the technology can analyze and interpret complex data sets, identify patterns, and make predictions based on the data. For example, in healthcare, machine learning can be used to analyze medical imaging data to aid in the early detection of diseases.

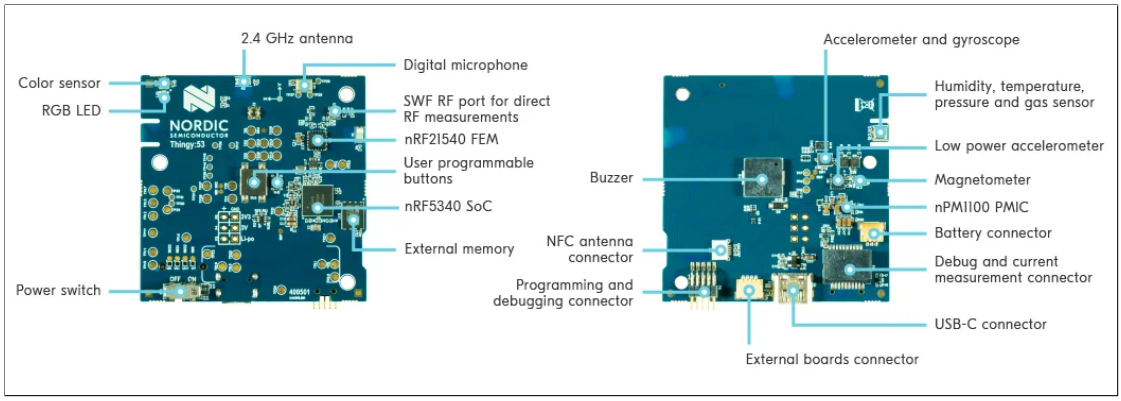


Figure 2.2: Nordic Thingy:53 Sensors and I/O Overview [1]

This technology is also used in smart home sensing. The system can be integrated with smart home devices to provide advanced sensing capabilities for environmental monitoring, energy management, and security. For instance, it can monitor air quality, detect motion, and optimize energy usage within a smart home environment.

Moreover, the technology is well-suited for fast prototyping in various industries. It enables rapid development of prototype products and systems, allowing engineers and designers to validate concepts and functionalities quickly. For instance, in the automotive industry, the technology can rapidly prototype and test new sensor-based features for vehicles.

Additionally, the technology is valuable for proof-of-concept development. It can be used to quickly build and test initial versions of new products or systems to assess feasibility and gather early feedback. For example, it can be employed in the consumer electronics industry to create proof-of-concept prototypes for innovative wearable devices or IoT gadgets.

2.3.3 nRF Edge Impulse machine learning app

Each Nordic Thingy:53 comes with pre-installed firmware for wirelessly transferring training data via Bluetooth LE to the cloud using the nRF Edge Impulse mobile app, allowing the creation of an embedded machine-learning model with Edge Impulse Studio. The model can then be deployed wirelessly to Thingy:53,

which handles inferencing, and the results can be viewed in the app. This functionality maximizes the potential of the advanced sensors of the Thingy:53 for applications such as voice recognition or movement pattern detection. Additionally, the low-power accelerometer and the PDM microphone can wake the SoC from sleep on motion or sound events, which is especially advantageous for developing low-power embedded machine-learning applications, enabling the device to conserve power during idle periods [1].

2.3.4 nRF Programmer app

The nRF Programmer app for the Thingy:53 introduces a new level of simplicity not previously seen in a prototyping platform of this calibre. With the app, you can select from pre-existing firmware for the Thingy:53 and wirelessly flash it from an iOS or Android device. This convenient feature enables you to upload new firmware to harness the capabilities of the nRF5340 SoC and the integrated sensors from anywhere without needing a PC connection [1].

2.3.5 nRF Connect SDK

The Thingy:53 is fully supported in the nRF Connect SDK for developing your firmware applications. This SDK is a scalable and unified software development kit for building products based on our nRF52, nRF53, and nRF91 Series wireless devices. It provides developers with an extensible framework for building firmware for devices and applications across all product categories, from simple to computationally intensive tasks. The SDK integrates the Zephyr RTOS with various samples, application protocols, protocol stacks, libraries, and hardware drivers. It offers a single code base for all our devices and software components, simplifying the process of porting modules, libraries, and drivers from one application to another and reducing development time. It ensures high memory efficiency by allowing developers to selectively choose essential software components for their applications [8] [1].

2.3.6 Specification

- **Application core:** This refers to the main processing unit of the device, which is a 128 MHz Arm Cortex-M33 with 1 MB Flash and 512 KB RAM. It handles the main application logic and computation.
- **Network core:** This is a 64 MHz Arm Cortex-M33 with 256 KB Flash and 64 KB RAM, dedicated to managing network-related tasks and communication protocols.
- **Power management IC (PMIC):** The nPM1100 PMIC manages the power supply and distribution within the device, ensuring efficient power usage and battery management.
- **RF front-end:** The device uses the nRF21540 FEM for its RF front-end, which is crucial for handling radio frequency signals and ensuring efficient wireless communication.
- **Wireless protocol support:** The device supports various wireless protocols, including Bluetooth LE, Bluetooth mesh, NFC, Thread, Zigbee, and 2.4 GHz proprietary protocols, enabling versatile connectivity options.
- **External connectors:** The device features Qwiik, Stemma, and Groove compatible 4-pin JST connectors, providing flexible options for external device connections.
- **Battery:** The device is powered by a Li-Po battery and supports USB-C rechargeability, ensuring convenient and portable power options.
- **LED:** The device incorporates an RGB LED that can be programmed to provide visual indicators or feedback.
- **Buttons:** The device has programmable buttons accessible through the casing, providing user input and interaction options.
- **Inertial measurement unit:** This includes a 6-axis accelerometer, providing accurate motion sensing capabilities.

- **Low power accelerometer:** The device's accelerometer can wake the device based on motion detection, contributing to power-efficient operation.

- **Environmental sensors:** The device is equipped with sensors for measuring temperature, humidity, air quality, and air pressure, providing environmental data for various applications.

- **Color and light sensor:** The device includes a sensor for detecting lighting conditions and colour information, enabling responsive adjustments based on ambient light.

- **Microphone:** The device features a PDM microphone capable of waking the device based on sound, offering audio input and control capabilities.

- **Buzzer:** The device includes a piezoelectric buzzer with a frequency of 4 kHz, providing auditory alerts or notifications when necessary.

2.4 DIME

DIME, short for DyWA Integrated Modeling Environment, is a tool for developing web applications using a model-driven approach. It is built on OTA and XMDD principles and emphasizes agile user-level software development and rapid prototyping. DIME employs a range of graphical domain-specific languages to address different facets of web applications, including data modelling, business logic, user interface, and access control [16].

The modelling environment in DIME enables high-level abstraction, focusing on describing what the application should achieve rather than how it is implemented. This supports prototype-driven development, where various graphical models represent different aspects of the target application. DIME aims to simplify development tasks, promote agility, and ensure quality assurance in application development [16].

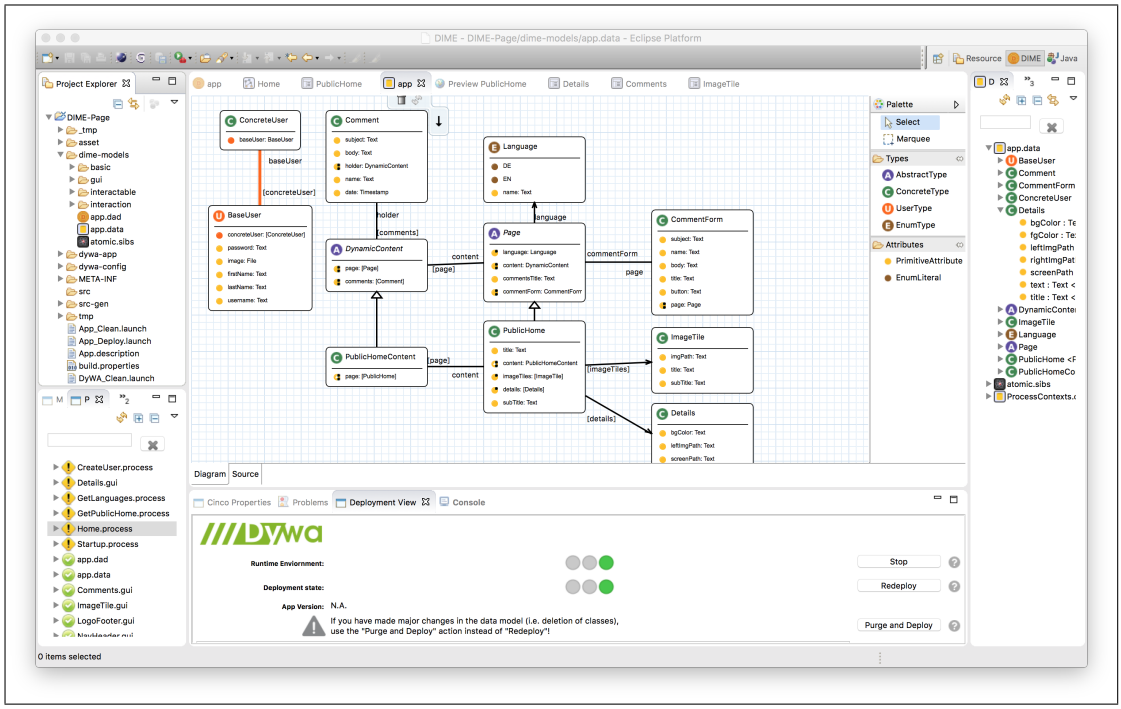


Figure 2.3: DIME Interface [2]

2.5 Integration of Thingy:53

In the Integration of Thingy:53 section, we will discuss how the Nordic Thingy:53 device is integrated into various platforms to enhance its functionality within our system. We will start by exploring the integration of Thingy:53 and MongoDB, where the device is connected to a database to store and retrieve sensor data efficiently. Next, we will delve into the integration of Thingy:53 and Pyrus, detailing how the device interfaces with task management software to streamline operations. Lastly, we will examine the integration of Thingy:53 and DIME, highlighting how this connection enables advanced data analytics and machine learning capabilities within the system. This section will outline the critical role of these integrations in achieving the project's objectives.

2.5.1 Integration of Thingy:53 and MongoDB

- **Data Collection**

Thingy:53 sensors are advanced devices that capture many data points, such as temperature, pressure, humidity, and air quality. These sensors initially store the collected data in CBOR format, a concise binary format for data interchange, and later convert it to JSON format to facilitate smoother processing. This conversion allows for seamless and efficient data handling, ensuring the information can be efficiently utilized for various applications and analyses [11].

- **Data Storage**

  MongoDB, a NoSQL database, is particularly adept at managing the diverse data generated by IoT devices. Its flexible schema and scalability make it an ideal choice for handling IoT data's dynamic and unstructured nature. MongoDB's BSON format allows data to be stored in key-value pairs, enabling efficient storage and retrieval of structured and unstructured data through queries. This format also supports complex data types, making it well-suited for representing the varied and evolving data structures often encountered in IoT applications. Overall, MongoDB's capabilities make it a robust and adaptable choice for managing the complexities of IoT data [11] [10].

- **Analytics Pipeline**

  The Pyrus analytics pipeline is a robust system designed to extract data from MongoDB, a NoSQL database, to derive valuable insights. By efficiently processing the data, Pyrus creates compelling visual representations, such as graphs and charts, that empower decision-makers to understand the information better. These visual aids enable stakeholders to make informed decisions based on the patterns and trends uncovered by the data analysis conducted through the Pyrus platform [11].

- **Digital Thread Layer**

  In the DIME application, domain-specific languages (DSLs) enable smooth communication and interaction among different tools. These DSLs streamline the exchange of information and enhance interoperability within the system. This capability is instrumental in creating a cohesive and efficient environment where tools can collaborate seamlessly to achieve common goals.

2.5.2 Integration of Thingy:53 and Pyrus

The Thingy:53 IoT package comes with the IoT sensor device and a dedicated app developed by the Thingy team. Users can tailor the sensor's behaviour and manage connectivity through protocols such as Bluetooth LE, WiFi, and Zigbee to gather data. To integrate Thingy:53 with Pyrus, the data collected by the Thingy app is stored in the MongoDB Atlas database. The Pyrus analytics pipeline retrieves this data, creating visual representations and insights to support informed decision-making and action. In the Digital Thread layer, data is fetched from MongoDB via REST API external DSLs, analyzed using the Pyrus pipeline, and displayed on a web application defined in DIME for stakeholders to visualize strategic information and make decisions [11].

2.5.3 Integration of Thingy:53 and DIME

Data from Thingy:53 is retrieved from MongoDB through REST API DSLs within the Digital Thread layer. The Pyrus pipeline processes the data, and a dashboard is generated in a web application defined in DIME for stakeholders to visualize information and facilitate decision-making. Building custom firmware is required to forward Thingy:53 data to another infrastructure. As a result of this limitation, collected measurements are currently uploaded to the database using a batch script. The entire system, encompassing Thingy:53, the smartphone app, Pyrus pipelines, and the DIME web app, effectively observed and logged changes in greenhouse parameters. The Pyrus pipelines produced a results dashboard integrated into the DIME dashboard to offer a user-friendly approach [11].

3

Design Process and Timeline

The Design Process and Timeline chapter outlines the structured approach to developing and completing the project, from the initial concept to the final submission. This chapter details the critical phases of the project, beginning with the formulation of the initial concept and requirements gathering through a review of related work and the implementation of the system, to the data analytics phase, where models are trained and evaluated. It also covers the writing and revising process, ensuring the project is well-documented and polished before submission. The chapter concludes with a Gantt chart, visually representing the timeline and milestones achieved throughout the project's duration.

3.1 Thesis Timeline

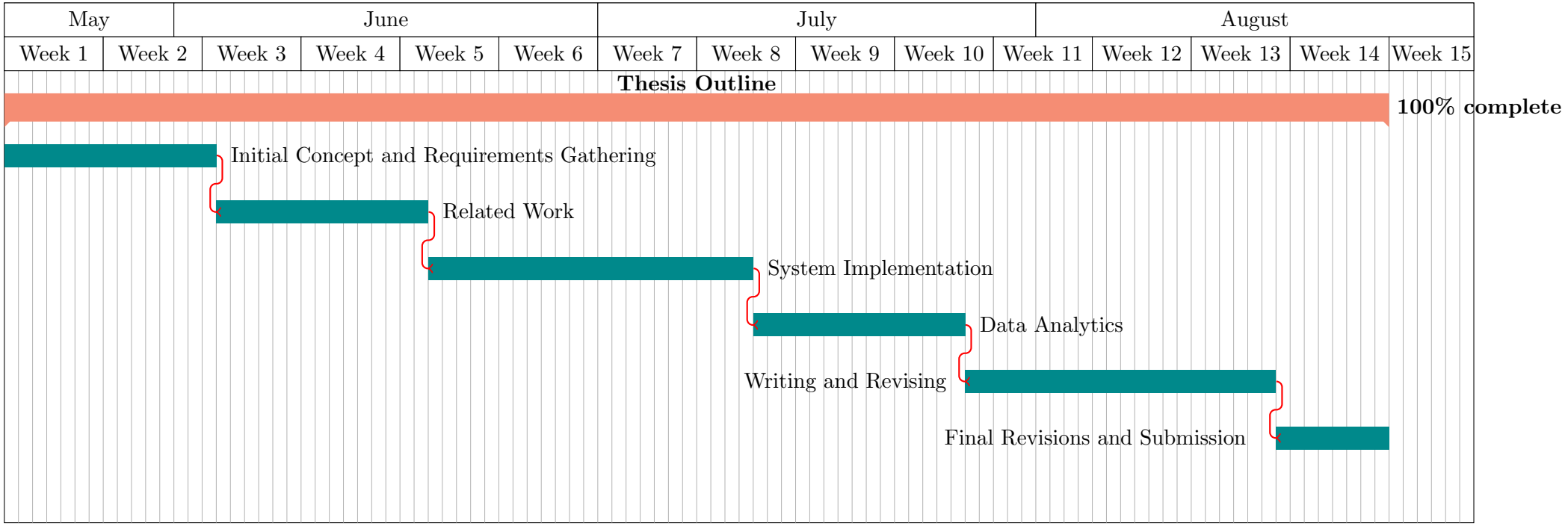


Table 3.1: Thesis Timeline

3.2 Initial Concept and Requirements Gathering

The Initial Concept and Requirements Gathering phase will occur from May 20 to June 3, 2024. The main goal will be to plan and outline the development of a prototype IoT-based system for real-time monitoring of the egg-hatching environment. This phase will be crucial in defining the project's scope and identifying the key technologies and methodologies to be used.

During this period, the focus will be on selecting the appropriate Platform and Development Environments to support integrating various sensors and processing the data. The Nordic Thingy:53 device will be chosen as the central hardware component due to its comprehensive suite of onboard sensors, including temperature, humidity, pressure, light, and air quality sensors. These sensors will be essential for accurately monitoring the hatching environment, which is critical for optimizing hatch rates.

Additionally, the project will emphasize the application of data analytics and machine learning methods to the data collected from these sensors. The goal will be to monitor and analyze the environmental conditions to predict and maintain optimal settings for egg incubation. This approach will involve collecting large datasets, training machine learning models, and deploying these models to make real-time decisions and adjust the hatching environment.

The requirements-gathering phase will also involve detailed discussions on integrating the hardware with the chosen software environment to ensure the system will be scalable and adaptable to future enhancements. By the end of this phase, a clear roadmap will be established, setting the stage for the subsequent stages of system design, development, and implementation.

3.2.1 System Architecture Diagram

add an architecture diagram showing the sensor layout, software connections, and data analytics part

Draft

An IoT-based prototype system for real-time monitoring of the egg-hatching environment. This objective was attained using the Nordic Semiconductor Thingy:53, a flexible IoT device with various sensors.

The full range of sensors in Thingy:53 was utilized to gather comprehensive environmental data. These sensors measure temperature, humidity, pressure, light, and air quality, all vital factors in the egg-hatching process.

Obtain sensor data from the Nordic Thingy:53, furnished with BME688 and BH1749 sensors, and deliver it in JSON format through an API request. An HTTP server will be established directly on the device. This server will manage incoming HTTP requests and reply with the requested sensor data formatted as JSON. The information gathered from these sensors offered a detailed understanding of the conditions within the incubation environment. This data was necessary for real-time monitoring, subsequent data analysis, and machine-learning tasks to improve the hatching conditions.

The API requests will be run from DIME (DyWA Integrated Modeling Environment), a tool designed for developing web applications using a model-driven approach.

The requested sensor data will be displayed in the DIME user interface, enhanced with graphs and diagrams, providing an intuitive and interactive way to visualize the environmental conditions. This data was saved in CSV format, enabling further analysis and implementation of machine learning applications.

During the data analytic workflows, we execute steps to construct and assess machine learning models based on sensor data to predict optimal conditions.

The process commences by importing the dataset, split into two phases: Setter and Hatcher. These sets are then split into train and test datasets in preparation for model training.

For this workflow, employ Python with Jupyter Notebook in Visual Studio Code, which allows for interactive analysis and visualisation of the data.

We have opted for Logistic Regression and Support Vector Machine (SVM) models for this analysis.

Both models are trained on the training dataset to predict whether the conditions are ideal or non-ideal.

After training the Model, we use it to predict the test dataset and evaluate its performance using accuracy, precision, recall, F1 score, and confusion matrix metrics.

Utilise the trained models to predict environmental conditions based on actual sensor data. The sensor data from device Thingy:53, with the onboard BME688 sensor, collects environmental data such as temperature and humidity. Using logistic regression and SVM, we aim to achieve reliable predictions for ensuring optimal conditions in different scenarios.

The results are saved in a CSV file for further analysis and reporting.

3.3 Related Work

The Related Work phase, scheduled from June 4 to June 18, 2024, will aim to study and understand upcoming technologies, methodologies, and applications that will shape the project's development. This study will be vital for establishing a robust theoretical foundation and identifying the most suitable technologies and approaches for building the prototype IoT-based egg-hatching monitoring system.

The first focus area will be the Adopted Technology, which will review the Development Environment, Internet of Things (IoT) frameworks, and Artificial Intelligence (AI) techniques relevant to the project. This will include exploring various development tools, platforms, and libraries to support the seamless integration of hardware and software components and enable efficient data processing and analysis. The review of IoT technologies will focus on collecting, transmitting, and utilizing sensor data in real-time monitoring systems. At the same time, exploring AI methods will aim to understand how machine learning models could optimize the egg-hatching process based on environmental data.

Another crucial area of study will be the application domain of egg-hatching. This will involve studying natural and artificial incubation methods, understanding the factors influencing hatchability, and reviewing existing systems and technologies used in poultry farming. Analyzing current practices' challenges and limitations will help identify improvement opportunities through IoT and AI technologies.

Much of this phase will be dedicated to the Nordic Thingy:53, the project's chosen hardware platform. The literature review will include an in-depth analysis of its features, capabilities, and previous use cases in similar IoT applications to understand its full potential and compatibility with various software tools.

Finally, the Integration of Thingy:53 will be studied to ensure an effective combination with other system components, such as databases, cloud platforms, and data analytics tools. The review will focus on the technical aspects of interfacing Thingy:53 with other systems, ensuring data flow continuity, and maintaining real-time monitoring capabilities. This study will provide the knowledge to design a robust and scalable system architecture supporting future enhancements and modifications.

3.4 System Implementation

During the System Implementation phase, scheduled from June 19 to July 11, 2024, the project will focus on translating the design concepts into a functional prototype. This phase will begin with setting up the necessary platform and development environments. We will configure and optimize various development tools and environments, such as Nordic Semiconductor's nRF Connect SDK, Visual Studio Code (VS Code), and Zephyr OS, to ensure seamless integration and compatibility with the hardware. This setup will lay the foundation for the development and testing of the system, ensuring that the tools used are well-suited to handle the complexities of the project.

Following the environment setup, we will experiment with the Nordic Semiconductor Thingy:53. This stage will involve hands-on testing and validation of Thingy:53's capabilities, mainly focusing on its onboard sensors, such as temperature, humidity, and air quality. We will conduct various experiments to fine-tune sensor configurations, optimize data collection processes, and ensure that Thingy:53 can reliably gather and transmit real-time environmental data. This experimentation will be crucial for understanding the device's limitations and capabilities, informing subsequent project stages.

The next step will involve reading sensor data via API requests. We will develop and implement APIs to effectively retrieve data from the Thingy:53 sensors over USB connections. This process will include writing code to make API requests, handle responses, and parse the sensor data into usable formats. The goal will be to establish a reliable data pipeline that can be used for continuous monitoring

and analysis. We will also ensure that the data retrieval process is efficient and can operate with minimal latency, critical for real-time monitoring applications.

Finally, we will move on to Building the Prototype. To efficiently obtain sensor data from the Nordic Thingy:53, furnished with BME688 and BH1749 sensors, and deliver it in JSON format through an API request, an HTTP server will be established directly on the device. This server will manage incoming HTTP requests and reply with the requested sensor data formatted as JSON. The API requests will be run from DIME (DyWA Integrated Modeling Environment), a tool designed for developing web applications using a model-driven approach. The requested sensor data will be displayed in the DIME user interface, enhanced with graphs and diagrams, providing an intuitive and interactive way to visualize the environmental conditions.

Moreover, the prototype will collect and store sensor data in CSV format, which will later be used for machine learning and data analytics. This capability will enable the application of advanced analytical techniques to the collected data, potentially leading to insights that can improve the egg-hatching process. This stage will involve integrating all the components—hardware, software, and data pipelines—into a cohesive system, culminating in a working prototype that demonstrates the feasibility of the IoT-based egg-hatching monitoring system.

3.5 Data Analytics

During the Data Analytics phase, scheduled from July 12 to July 26, 2024, our project will focus on building and evaluating machine learning models to predict the optimum conditions for egg incubation based on sensor data collected from the Nordic Thingy:53. The workflow will start with importing the dataset, which is divided into two important phases: Setter and Hatcher. These datasets will be split into train and test datasets to ensure the models are well-prepared for the training process. This division is essential for assessing the model's ability to generalize from the training data to new, unseen data.

We will use Python in Jupyter Notebook integrated into Visual Studio Code in this analytical workflow. This setup will create an interactive environment for data

exploration, enabling real-time analysis and visualization. The advanced visualization capabilities will help us understand data distributions, identify patterns, and spot potential anomalies that could impact model performance.

We will focus on training two primary machine learning models: Logistic Regression and Support Vector Machine (SVM). These models are chosen for their robustness and effective performance in classification tasks, essential for predicting whether the environmental conditions are optimal for each egg incubation phase. The training process will involve fine-tuning the models to achieve the best possible performance, followed by predicting labels for the test data. This prediction step will demonstrate how well the models can classify the conditions based on the sensor data.

After training the models and making predictions, the next step is to evaluate their performance thoroughly. We will assess the Logistic Regression and SVM models' accuracy, precision, recall, and F1 scores. This will provide insights into their effectiveness and help identify areas for improvement. The evaluation is crucial for determining the reliability of the predictions and ensuring that the models are suitable for real-world deployment.

The workflow will include a step to predict sensor data conditions. This process will use the trained models to make real-time predictions based on new sensor data, providing actionable insights into the incubation environment's status. This capability will be a key outcome of the data analytics phase, enabling proactive adjustments to the incubation process to maximize hatchability and overall success.

3.6 Writing and Revising

During the Writing and Revising phase, scheduled from July 27 to August 17, 2024, we will document the entire project process and refine the final deliverables. We will use this time to write comprehensive sections for the project report detailing the design, implementation, data analytics, and outcomes of the IoT-based egg-hatching monitoring system. Each chapter will be written meticulously to ensure clarity, technical accuracy, and coherence, highlighting the innovative approaches and results.

In addition to writing, we will engage in rigorous revision cycles. This will involve reviewing the content for grammatical precision, consistency in terminology, and alignment with the project's goals and objectives. Feedback from the supervisors will be integrated to enhance the quality and depth of the analysis presented in the report. Special attention will be given to the clarity of complex technical sections, ensuring accessibility to a broad audience, including stakeholders with varying technical expertise.

Furthermore, this phase will include preparing visual aids, such as figures, source code snippets, diagrams, charts, and tables, to effectively communicate the project's findings. These visuals will be aligned with the text to reinforce key points and provide readers with a clear understanding of the data and results.

The final output of this phase will be a polished, professional document ready for submission. It will reflect the project's comprehensive scope, from initial concept to prototype development and data analytics, serving as both a technical reference and a showcase of its achievements.

3.7 Final Revisions and Submission

The Final Revisions and Submission phase, scheduled from August 18 to August 25, 2024, represents the concluding stage of the project. During this period, we will review the entire project documentation thoroughly. The goal is to ensure that all sections are carefully polished and error-free. This phase will involve conducting final checks for technical accuracy, ensuring consistency across all chapters, and adhering to the project's goals and guidelines.

We will also integrate any last-minute feedback received from supervisors. The revisions made during this phase will enhance the document's clarity, coherence, and overall professionalism. Additionally, we will verify that all data and references are correctly cited. Furthermore, visual elements such as figures, source code snippets, graphs, charts, and diagrams will be accurately labelled and effectively integrated into the text.

In addition to revising the document, We will prepare it for submission to ensure it meets all required formatting and submission guidelines. This may include converting the document into the required format (PDF) and double-checking that

all appendices, including code, data, and supplementary materials, are correctly included and accessible.

The conclusion of this phase will be the official submission of the final project report. We will also document the submission process, ensuring all necessary forms and documentation are completed and submitted alongside the report. This marks the successful completion of the project, with all milestones achieved and documented, paving the way for the project's formal evaluation and potential future application or publication.

4

System Implementation

The chapter on System Implementation provides detailed steps for developing an IoT-based egg-hatching monitoring system using the Nordic Thingy:53 device. It starts by outlining the foundational Platform and Development Environments. The following section, Install and Setup DIME, explains how to set up the Data Intelligence Machine Learning Environment (DIME) for advanced data analysis. Next, Experimentation with Nordic Semiconductor Thingy:53 describes the initial experiments conducted with the device. The section Reading Sensor Data via API Request over USB explains how sensor data is collected from Thingy:53 via USB.

4.1 Platform and Development Environments

The creation of the IoT-based egg-hatching monitoring system was made possible with the support of the Nordic Semiconductor Thingy:53 as the primary hardware platform. The firmware was developed using the nRF Connect SDK, with essential tools like the nRF Command Line Tools and nRF Connect for Desktop being used for device management. The integrated development environment was Visual Studio Code (VS Code), and the underlying operating system for Thingy:53 was provided by Zephyr OS. DIME was employed for data integration and exploration, while sensor data was stored and managed using MongoDB. Here's an explanation of each component.

4.1.1 Nordic Semiconductor Thingy:53

This state-of-the-art IoT prototyping platform comes with various sensors for monitoring environmental conditions. The Thingy:53 is compatible with Bluetooth Low Energy (BLE) and is powered by the nRF5340 SoC, making it perfect for our egg hatch optimisation project. It enables real-time data collection and transmission [1]. Nordic Semiconductor provides a comprehensive software development kit containing libraries, example code, and development tools. These tools aim to streamline the process of creating IoT applications. The kit is compatible with various connectivity protocols and can be integrated with other development tools.

4.1.2 nRF Connect SDK

The nRF Connect SDK is a flexible and cohesive software development kit designed for constructing low-power wireless applications using Nordic Semiconductor's nRF52, nRF53, nRF70, and nRF91 Series wireless devices. It provides a customisable framework for developing compact software for memory-limited devices and robust and intricate software for more advanced devices and applications. The components of the nRF Connect SDK include the nRF Connect for VS Code Extension Pack and Toolchain. The nRF Connect for VS Code extension pack allows developers to use the popular Visual Studio Code Integrated Development Environment (VS Code IDE) for creating, compiling, debugging, and deploying embedded applications built on Nordic's nRF Connect SDK. The toolchain includes various tools for building applications in the nRF Connect SDK, such as the assembler, compiler, linker, and CMake [3].

4.1.3 nRF Command Line Tools

The nRF Command Line Tools is a versatile software suite for cross-platform desktop use. It is primarily employed for developing, programming, and debugging Nordic Semiconductor's nRF51, nRF52, nRF53, and nRF91 Series devices. The components of this toolset include the nrfjprog executable, which facilitates programming through SEGGER J-LINK programmers and debuggers, and the mergehex executable, allowing the combination of up to three .HEX files into a

single file, and the nrfjprog DLL—a dynamic-link library that exports functions for programming and controlling nRF51, nRF52, nRF53, and nRF91 Series devices. Additionally, this DLL empowers developers to create their development tools using the DLL's API. The software and documentation pack for the SEGGER J-Link is also included in this comprehensive package. One of the notable aspects of the nRF Command Line Tools is its compatibility with multiple platforms, including Windows, Linux, and macOS. This cross-platform support ensures that developers and engineers can seamlessly utilise the tools across different operating systems, enhancing their flexibility and productivity in working with Nordic Semiconductor's devices [17].

4.1.4 nRF Connect for Desktop

nRF Connect for Desktop is a comprehensive software toolset designed to facilitate the development of Nordic Products across different platforms [18]. It offers a wide range of applications to test, monitor, measure, optimise, and program nRF devices, making it a versatile and essential tool for developers. One of its key features is the ability to identify the development kits and dongles connected to the computer and then upload the necessary firmware for these devices. This ensures seamless integration with the development environment and streamlines the firmware update process. Please remember the nRF Connect for Desktop applications includes Bluetooth Low Energy, Board Configurator, Cellular Monitor, Direct Test Mode, nPM PowerUP, Power Profiler, Programmer, Quick Start, RSSI Viewer, Serial Terminal, and Toolchain Manager. The most commonly used apps are Programmer 7.52 and Serial Terminal7.53.

4.1.5 Visual Studio Code (VS Code)

VS Code 7.54, a source code editor with open-source support, accommodates various programming languages and development tools. Its versatility and extensive range of extensions make it an excellent platform for coding, debugging, and version control in IoT development. Visual Studio Code, a lightweight yet robust source code editor, is compatible with Windows, macOS, and Linux operating systems. It offers built-in support for JavaScript, TypeScript, and Node.js and boasts

a diverse selection of extensions for other languages and runtimes, such as C++, C#, Java, Python, PHP, Go, and .NET [7].

4.1.6 Zephyr OS

Zephyr OS is a real-time operating system designed for devices with limited resources supported by the nRF Connect SDK. It is tailored to develop secure and reliable IoT solutions essential for maintaining precise control in an egg-hatch environment. The Zephyr Project, hosted by the Linux Foundation, is a collaborative open-source effort that brings together developers and users to build a top-notch small, scalable, real-time operating system (RTOS) optimised for resource-constrained devices across multiple architectures.

The Zephyr Project provides a neutral platform for silicon vendors, OEMs, ODMs, ISVs, and OSVs to contribute technology, reducing costs and speeding up time to market for billions of connected embedded devices. The software is an excellent option for simple connected sensors, LED wearables, modems, and small wireless gateways. Thanks to its modularity and support for multiple architectures, developers can create solutions that fit their requirements.

Being an open-source project, the community continually enhances the project to support new hardware, developer tools, sensors, and device drivers. Updates are regularly rolled out to integrate improvements in security, device management capabilities, connectivity stacks, and file systems [9].

4.1.7 DIME (DyWA Integrated Modeling Environment)

DIME offers a model-driven approach for building web applications. It focuses on agile development and quick prototyping through graphical domain-specific languages. These languages represent data, business logic, user interfaces, and access control. DIME's high-level abstraction streamlines development processes and guarantees quality assurance [2].

4.1.8 MongoDB

A database that is not based on SQL and offers adaptable and expandable data storage, MongoDB is especially suitable for managing the varied and ever-changing data produced by IoT devices. It facilitates the effective storing and fetching of sensor data, which aids in real-time analysis and decision-making [10] [11].

4.2 Install and Setup DIME

The following is a systematic guide for installing DIME. By adhering to these instructions, we will establish DIME on our system, preparing it for developing web applications using an approach grounded in model-driven principles [2].

4.2.1 Download the Installer

To download the installer for the application, we have two options: Daily Builds and Latest Builds. For daily builds that are updated regularly, we can download the installer for Linux, MacOS, and Windows from the link:
`https://ls5download.cs.tu-dortmund.de/dime/daily/`
And for the Latest Builds, we can choose to Download for Linux with the link:
`https://ls5download.cs.tu-dortmund.de/dime/daily/dime-latest-linux.zip`
Download for MacOS with the link:
`https://ls5download.cs.tu-dortmund.de/dime/daily/dime-latest-macosx.zip`
Download for Windows with the link:
`https://ls5download.cs.tu-dortmund.de/dime/daily/dime-latest-win32.zip`

This way, we can select the specific build corresponding to our operating system. Please remember to confirm whether Java 11 has been installed on your computer. If it hasn't been installed, you can obtain it from Oracle Java SE 11 Archive Downloads with the link below and install it on your computer.
`https://www.oracle.com/ie/java/technologies/javase/jdk11-archive-downloads.html`

4.2.2 Run DIME

Depending on our operating system, the steps for running DIME are as follows: Using Linux or MacOS, we can run DIME from the terminal by executing the appropriate command. On Windows, we can run dime.exe by double-clicking the executable file. This will launch the dime application and allow us to use its features.

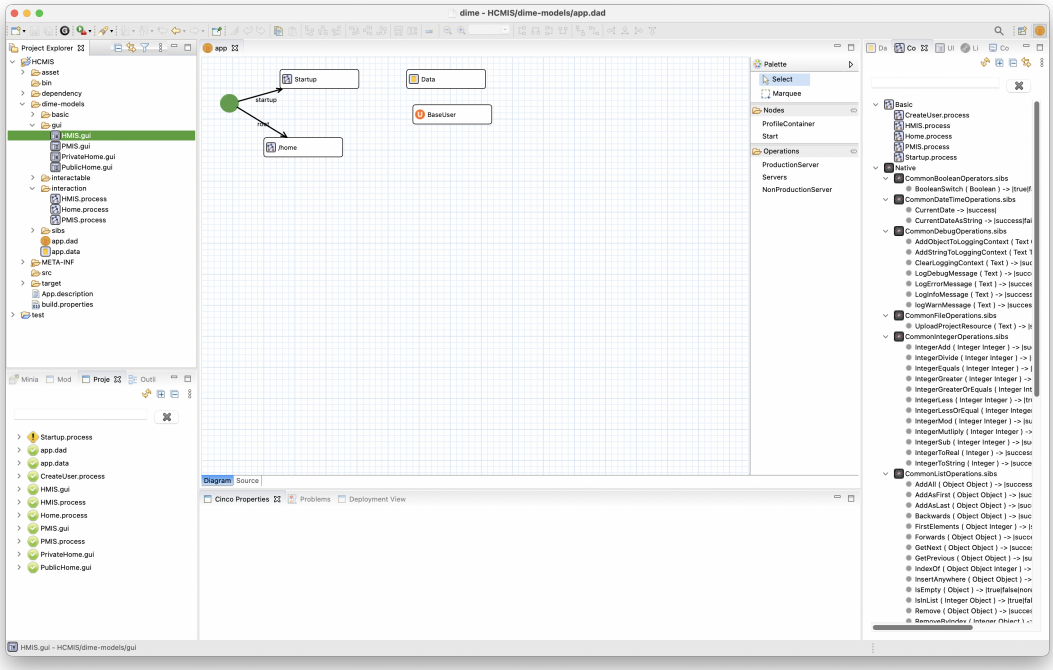


Figure 4.1: Running the DIME

4.3 Experimentation with Nordic Semiconductor Thingy:53

In this part, we explore the practical testing done with the Nordic Semiconductor Thingy:53, specifically focusing on its built-in sensors. The Thingy:53 comes with various environmental sensors, making it well-suited for real-time monitoring and data collection in an environment where eggs are hatching. The sections below

detail the specific tests carried out to assess the performance of these sensors, covering their connections, data collection methods, and simultaneous operations. Each test evaluates the accuracy and consistency of Thingy:53's internal sensors in capturing crucial environmental variables essential for improving the egg-hatching process. The sections include Sensors Connections; Read BME688 Sensor: Temperature, Humidity, Pressure, and Gas; Read BH1749: Digital Colour Sensor; and Read BH1749 and BME688 Sensors, with an explanation provided for each subsection.

4.3.1 Sensors Connections

Nordic Semiconductor Thingy:53 has various sensors, including microphones, light, accelerometers, magnetometers, buzzers, and gas sensors. These sensors are valuable for quickly developing machine learning applications. The sensor data can be utilised to train a machine-learning model, which can then be programmed into the board. Thingy:53 is compatible with edge impulse and allows for rapid training and deployment of machine learning models [19].

4.3.2 Essential Source Code Files for the Project Thingy:53

These files are essential for ensuring that Thingy:53 is set up and assembled and functions according to the precise project specifications, from initialising the hardware to running the primary application logic. Below is an explanation of each of these files' functions.

- `Kconfig`

  The `Kconfig` file plays a vital role in setting up the build environment for the Thingy:53 project. It enables developers to specify various configuration options and dependencies necessary for the project. By indicating which modules and features to enable, `Kconfig` guarantees that the firmware is tailored to meet the application's specific requirements. For Thingy:53, this file is essential for managing the configuration of sensors, communication interfaces, and other hardware components. It assists in customising the firmware to make the most of Thingy:53's specific capabilities.

- `prj.conf`

  The `prj.conf` file contains specific configurations that determine the behaviour and capabilities of the Thingy:53 firmware. It enables functionalities such as Bluetooth, sensor modules, and low-power modes, which are crucial for IoT applications. This file directly impacts the operation of Thingy:53 in a particular setting. It sets up the required drivers and middleware to ensure the device's proper functioning, such as setting up the temperature and humidity sensors for real-time data collection.

- `app.overlay`

  The file `app.overlay` is utilised to change the default device tree, a hardware abstraction layer in Zephyr OS that Thingy:53 employs. This file enables developers to adjust hardware configurations, such as GPIO settings or sensor connections, to ensure that the firmware interacts correctly with the hardware of Thingy:53. It is crucial for tailoring how Thingy:53 communicates with its hardware components. For instance, if a particular sensor is linked to a non-default pin, the `app.overlay` file can be used to reconfigure this without changing the core device tree.

- `CMakeLists.txt`

  The `CMakeLists.txt` file forms a crucial part of the CMake build system and guides the build process. It specifies including source files, libraries, and dependencies while compiling the Thingy:53 firmware. This file plays a vital role in ensuring the correct compilation of the firmware. It guarantees the inclusion and linking of all required components, such as sensor libraries and communication protocols, enabling Thingy:53 to operate as an integrated IoT device.

- `main.c`

  The primary logic for the application is found in the `main.c` file. This is where the fundamental functionalities, such as initialising sensors, processing data, and implementing communication protocols, are included. It's essentially the central component of the firmware, determining Thingy:53's

behaviour in its operation environment. For Thingy:53, `main.c` is where you develop the code that communicates with the device's sensors, gathers environmental data, and transmits it to other systems or stores it. This file outlines the real-time operations that enable Thingy:53 to function effectively as an IoT device for monitoring and managing the hatching environment.

The hierarchy of files in the system for a Thingy:53 project

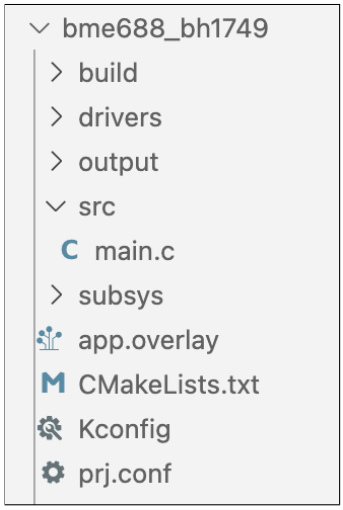


Figure 4.2: The Hierarchy of Essential Source Code Files

4.3.3 Read BME688 Sensor: Temperature, Humidity, Pressure, and Gas

The initial one is the Bosch Sensortech BME688 sensor. This sensor is the first of its kind capable of measuring outdoor gas, as well as temperature, humidity, and pressure [20]. It is compatible with I2C and SPI communication protocols. The sensor has AI capabilities; customisation can be done using BME studio for specific applications. It can detect volatile organic compounds (VOCs), volatile sulfur compounds (VSCs), and other gases like carbon monoxide and hydrogen within the parts per billion (ppb) range. The BME688 sensor is the first gas sensor with integrated high-linearity and high-accuracy pressure, humidity, and temperature sensors, and it is AI-capable [21]. Encased in a sturdy yet compact

3.0 x 3.0 x 0.9 mm^3 package, it is specially designed for mobile and connected applications where size and low power consumption are crucial. This gas sensor can detect VOCs, VSCs, and other gases, such as carbon monoxide and hydrogen, within the part per billion (ppb) range [22] [23]. A detailed explanation is already provided in the Appendix for the Source Code 7.2 files (prj.conf 7.1, app.overlay 7.2, CMakeLists.txt 7.3, main.c 7.4) Output 7.2 and Data Interpretation 7.2.

4.3.4 Read BH1749: Digital Colour Sensor

The BH1749 digital colour sensor is an advanced device specifically engineered to detect and convert Red, Blue, Green, and Infrared wavelengths into precise digital representations. This allows for highly accurate measurement of ambient light illuminance and colour temperature. The sensor is specifically designed to be compatible with the I2C bus interface, ensuring smooth and seamless operation within a variety of systems and applications [24] [25] [3]. A detailed explanation is already provided in the Appendix for the Source Code 7.2 files (Kconfig 7.7, prj.conf 7.9, app.overlay 7.8, CMakeLists.txt 7.10, main.c 7.11) Output 7.2 and Data Interpretation 7.2.

4.3.5 Read BH1749 and BME688 Sensors

The provided data gives a comprehensive overview of the combined output from the onboard sensors assigned to plate numbers 1 and 2. This comprehensive dataset includes detailed temperature, pressure, humidity, and gas resistance readings. Additionally, it provides specific measurements for red, green, blue, and infrared parameters [22] [23]. A detailed explanation is already provided in the Appendix for the Source Code 7.2 files (Kconfig 7.17, prj.conf 7.20, app.overlay 7.19, CMakeLists.txt 7.21, main.c 7.22) Output 7.2, Data Interpretation 7.2 and Implications for the Egg-Hatching Environment 7.2.

4.4 Reading Sensor Data via API Request over USB

To efficiently obtain sensor data from the Nordic Thingy:53, furnished with BME688 and BH1749 sensors, and deliver it in JSON format through an API request, we must establish an HTTP server directly on the device. This server will manage incoming HTTP requests and reply with the requested sensor data formatted as JSON. The Nordic Thingy:53 does not have built-in Wi-Fi hardware, so communication must be conducted via a USB connection. This necessitates integrating USB functionality into the firmware and setting up the device as a network interface over USB. Once this setup is complete, the device can accommodate API requests, allowing efficient access to sensor readings over a direct USB connection.

4.4.1 Sensor Configuration and Initialisation

Initially, the sensors on the Thingy:53 undergo configuration and initialisation. This involves enabling the appropriate drivers and ensuring the correct implementation of the initialisation code for sensors such as BME688 and BH1749. Accurate sensor readings and subsequent data processing rely on proper initialisation. The BME688 sensor gathers environmental data, including temperature, humidity, pressure, and air quality (measured via gas resistance). In contrast, the BH1749 sensor measures light levels and captures the red, green, blue, and infrared (IR) components. Collecting this data is a crucial first step in preparing it for API responses.

In previous experimentation detailed in the section titled "Experimentation with Nordic Semiconductor Thingy:53" 4.3 under subsections such as "Reading BME688 Sensor: Temperature, Humidity, Pressure, and Gas" 4.3.3, "Reading BH1749: Digital Colour Sensor" 4.3.4, and "Reading BH1749 and BME688 Sensors" 4.3.5, we illustrated the process of sensor configuration, initialisation, and data reading from the Thingy:53 device. The sensors successfully read data and output the results to a serial terminal. This data was later converted into JSON format, making it accessible via API requests to the HTTP server's JSON endpoint.

For real-time monitoring, thresholds can be set for critical sensor values, such as temperature or humidity. If these thresholds are exceeded, the system can trigger real-time alerts. Depending on the application's needs, these alerts can notify users or initiate automated responses, enhancing the system's ability to effectively manage and respond to critical conditions.

4.4.2 Setting Up the USB Network Interface

Configuring the Thingy:53 as a USB network adapter requires the USB Communications Device Class Abstract Control Model (CDC ACM) class to establish a virtual serial port over USB alongside Ethernet-based networking support. The system can manage IPv4 and IPv6 protocols, with active DHCP to allocate IP addresses dynamically. This setup allows the Thingy:53 to function as a network interface over USB, facilitating API requests and data communication with a host computer. Initiate the USB functionality and set up the network interface to ensure a stable connection over USB. This is crucial to guarantee effective communication between the device and the host system.

Once the appropriate firmware is flashed onto the Thingy:53, the host computer should recognise it as a USB network adapter. Verify that the host computer identifies Thingy:53 as a new network interface upon connection. With DHCP enabled, the device should automatically obtain an IP address, enabling immediate network communication. A static IP address can be manually assigned based on the specific network configuration if DHCP is unavailable or a fixed network setup is preferred. This flexibility allows seamless integration of the Thingy:53 into various network setups, making it a reliable tool for network-based applications and API interactions. A detailed explanation is already provided in the Appendix for the Source Code 7.2 files (prj.conf 7.30, app.overlay 7.31, CMakeLists.txt 7.32, main.c 7.33) and Output 7.2.

Throughout our development process, we faced significant obstacles while setting up the USB network interface on the Nordic Thingy:53 using the nRF Connect SDK. We could not establish a stable network connection despite thorough troubleshooting and multiple configuration attempts, such as configuring the device as a USB CDC ACM (Communication Device Class Abstract Control Model).

The main issue was the inability to properly initialise the default network interface, which hindered the device from obtaining an IP address and establishing a reliable connection. Due to time constraints, we decided to set this issue aside temporarily. We will explore alternative solutions or revisit this problem with a fresh perspective later. Our current focus will shift to other priorities.

4.4.3 Setting Up the USB HTTP Server and JSON API Endpoint

To configure a Zephyr-based device, such as the Nordic Thingy:53, to operate as an HTTP server, it is necessary to set up the device to manage network communications over USB and handle incoming HTTP requests. The HTTP server should be able to process specific API requests, listen to incoming requests, and produce appropriate responses based on the requested endpoints, including JSON-formatted data. Initially, the device should be configured as a USB network adapter using the USB CDC ACM (Communications Device Class Abstract Control Model). This configuration allows the host computer to recognise the device as a network interface when connected via USB. The USB device should be set up as a network interface using Ethernet over USB by initialising the network interface in the `main.c` file of the application, as previously explained. Once the network interface is operational, the next step is to initiate an HTTP server on the device. This server will listen to incoming HTTP requests and provide relevant data in JSON format, such as web pages or sensor data. Zephyr RTOS offers a built-in HTTP server that can be utilised for this purpose. To implement this, ensure the necessary headers are included, and the HTTP server commences within the application code.

An essential aspect of this setup is creating a JSON API endpoint. This endpoint will enable clients to retrieve sensor data in JSON format. For example, it might involve defining a function that collects sensor readings and formats them into a JSON object. This data can then be sent as a response to a specific API request. After developing the server and API, compile and flash the firmware onto the Thingy:53 device. The device should function as a network interface on the host computer when connected via USB. The device will obtain an IP

address through DHCP or a predefined static IP if the configuration is correct. Consequently, the HTTP server and its JSON API endpoints will be accessible by entering the device's IP address into a web browser or utilising command-line tools such as `curl`.

For example, to retrieve sensor data in JSON format, we may issue a command similar to this:

```
curl http://<device-ip>/api/sensors
```

This command should return the sensor data formatted as JSON, providing a simple and effective way to interact with Thingy:53's sensor outputs via the USB-connected network interface. A detailed explanation is already provided in the Appendix for the Source Code 7.2 files (Kconfig 7.41, prj.conf 7.42, CMakeLists.txt 7.44, main.c 7.45) and Output 7.2.

We faced many build errors when we tried to set up the USB HTTP server and JSON API endpoint on the Nordic Thingy:53. These problems varied from conflicts in configuration to unresolved dependencies within the nRF Connect SDK. Despite our attempts to troubleshoot and modify various settings, the build errors persisted, hindering us from establishing a functional setup. Due to time constraints and the complexity of the issues, we have opted to postpone this task temporarily. We intend to explore alternative solutions or return to the problem with a fresh perspective. This approach will allow us to concentrate on other critical priorities while ensuring we can return to this challenge with a clearer understanding and a better-prepared strategy.

4.4.4 Prototype Solution

Meanwhile, we will gather sensor data from Thingy:53 for the Prototype Solution and store it in CSV format. This information will be utilised for machine learning purposes, specifically for training a classification model with the help of ideal egg hatch data. We will refine and evaluate classification models such as the Logistic Regression and Support Vector Machine (SVM). These models will be trained to make predictions based on the gathered sensor data, allowing us to progress in

our research while preparing for the improved network capabilities offered by the nRF7002 EB.

5

Data Analytics Workflows

During the data analytic workflows, we execute steps to construct and assess machine learning models to predict optimal conditions based on sensor data. The process commences by importing the dataset, split into two phases: Setter and Hatcher. These sets are then split into train and test datasets in preparation for model training. We employ Python with Jupyter Notebook in Visual Studio Code for this workflow, allowing for interactive analysis and visualisation of the data.

We have opted for Logistic Regression and Support Vector Machine (SVM) models for this analysis. Logistic Regression is chosen for its simplicity and efficiency in binary classification tasks, while SVM is selected for its effectiveness in managing complex datasets and identifying optimal decision boundaries. Both models are trained on the training dataset to predict whether the conditions are ideal or non-ideal.

After training the Model, we use it to predict the test dataset and evaluate its performance using accuracy, precision, recall, F1 score, and confusion matrix metrics. These assessments help us understand how effectively the models work with new data. Then, we utilise the trained models to predict environmental conditions based on actual sensor data. The sensor data from device Thingy:53, with the onboard BME688 sensor, collects environmental data such as temperature and humidity. The results are saved in a CSV file for further analysis and reporting. This approach offers insights into our models' predictive accuracy and helps pinpoint potential areas for improvement. We aim to achieve reliable predictions for ensuring optimal conditions in different scenarios using Logistic Regression and

SVM. Below is a step-by-step explanation of the Machine Learning and Analytical process.

5.1 Install Packages

This arrangement guarantees that the essential libraries are set up for data analysis and machine learning activities within the Jupyter Notebook platform. The code consists of instructions for installing Python packages within the Jupyter Notebook framework.

```
%pip install --upgrade pip
%pip install matplotlib
%pip install numpy
%pip install pandas
%pip install seaborn
%pip install scikit-learn

# %pip will install the package in the virtual environment
# !pip will install the package in the base environment
```

Figure 5.1: Install Packages

`%pip install --upgrade pip` The following instructions update the `pip` package manager to the most recent version in the virtual environment linked with the Jupyter Notebook. `%pip install matplotlib` Installs the `matplotlib` package, enabling the creation of static, animated, and interactive visualisations in Python. `%pip install numpy` The `numpy` library is essential for numerical computing in Python, particularly for handling arrays and matrices. It is installed to facilitate these operations. `%pip install pandas` Installs the `pandas` package, which offers data structures such as DataFrames for manipulating and analysing data. `%pip install seaborn` Installs the `seaborn` package, a statistical data visualisation library built on top of `matplotlib` that offers a convenient interface for creating visually appealing and insightful plots. `%pip install scikit-learn` The `scikit-learn` library provides tools for machine learning such as classification, Regression, clustering, and dimensionality reduction. `%pip` vs `!pip`, `%pip` is

utilised for installing packages within the virtual environment associated with the Jupyter Notebook, ensuring that compatibility is maintained within that particular environment. Conversely, using `!pip` would result in the installation of packages in the base environment, potentially impacting global Python installations rather than the virtual environment associated with the notebook.

5.2 Import Libraries

The code is an essential part of a Python script or Jupyter Notebook, as it contains the necessary libraries and functions to facilitate tasks associated with data management, machine learning, and visualisation. Here's a detailed explanation.

```
# Computing
import numpy as np                                  # NumPy is fundamental for numerical computing in Python

# Dataset Handling
import pandas as pd                                 # Pandas is essential for data manipulation and analysis
#from pandas.api.types import CategoricalDtype      # This is used to define a categorical data type in a DataFrame

# Models and Machine Learning
from sklearn.model_selection import train_test_split # This function is used to split a dataset into training and testing sets
#from sklearn.model_selection import KFold          # splits the data into K subsets to validate the model's performance
#from sklearn.model_selection import cross_val_score # computes the cross-validated scores for a given estimator
#from xgboost import XGBRegressor                   # implementation of gradient boosted decision trees for regression
#from xgboost import XGBClassifier                  # implementation of gradient-boosted decision trees for classification
from sklearn.linear_model import LogisticRegression # linear model for binary classification
from sklearn.svm import SVC                         # model for classification tasks, particularly effective in high-dimensional spaces.
# evaluation metrics for assessing the performance of machine learning models
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix

# Plotting and Visualization
import matplotlib.pyplot as plt                     # creating static, animated, and interactive visualizations
import seaborn as sns                               # statistical data visualization library based on matplotlib
#from IPython.display import display                # display objects in Jupyter notebooks

# Miscellaneous
import warnings                                     # control the display of warning messages, allowing the user to ignore, display, or escalate
#from time import time, gmtime, strftime            # Imports specific functions from the time module
import os                                           # using operating system-dependent functionality
from pathlib import Path                            # object-oriented filesystem path manipulation library
from datetime import datetime                       # provides classes for manipulating dates and times in both simple and complex ways
```

Figure 5.2: Import Libraries

split image

- **Computing**

  NumPy, or `numpy`, is essential for conducting numerical computations in Python. It is handy for manipulating arrays and matrices and executing mathematical operations efficiently.

- **Dataset Handling**

  Pandas (`pandas`) are essential for manipulating and analysing data. They offer data structures such as Series (1D) and DataFrame (2D), essential for managing and working with datasets.

- **Models and Machine Learning**

  Train-Test Split (`train_test_split`): This function, available in `scikit-learn`, divides a dataset into training and testing sets. It is crucial for evaluating the performance of machine learning models. Logistic Regression

(`LogisticRegression`): It is a linear model used for binary classification and is employed to predict the likelihood of a binary outcome based on one or more features. Support Vector Classifier (`SVC`): This machine learning model in `scikit-learn` is utilised for classification tasks and shows particular effectiveness in high-dimensional spaces. Metrics (`accuracy_score`, `precision_score`, `recall_score`, `f1_score`, `classification_report`, `confusion_matrix`) These are various assessment measures used to evaluate the performance of machine learning models. They include accuracy, precision, recall, F1 score, classification reports, and confusion matrices.

- **Plotting and Visualization**

  Matplotlib (`matplotlib.pyplot`) is a Python plotting library that generates static, animated, and interactive visualisations. Seaborn (`seaborn`) is a statistical data visualisation library built on Matplotlib. It delivers a high-level interface for creating attractive and informative graphics.

- **Miscellaneous**

  The display of warning messages is controlled by the `warnings` module, allowing you to adjust, disregard, or escalate them as necessary. System-dependent functionality such as file manipulation, environment variables, and directory operations can be accessed using the `os` module. `pathlib.Path` from the Pathlib library offers an object-oriented approach to working with file paths for filesystem path manipulation. Manipulating dates and times in simple and complex ways is facilitated by the classes in the `datetime` module.

- **Purpose of Each Import**

  NumPy and Pandas are necessary for managing and processing data, which is the foundation of the data analytics process. The imports from `scikit-learn` are essential for creating, training, and assessing machine learning models. Matplotlib and Seaborn visualise data, providing better insight into the data and model performance. These imports manage warning messages, time functions, file operations, and date-time manipulations, all of which are crucial for the efficient execution and organisation of the code.

5.3 Configuration

The code is a setup block configuring different settings for plotting, handling warnings, and managing file directories. These initial steps guarantee the following code execution is efficient, with easily understandable visualisations and a well-structured file layout for managing data and outcomes. Please refer to the following explanation for each segment.

```python
# Set Matplotlib defaults
plt.style.use("seaborn-v0_8-whitegrid")          # Uses Seaborn's theme with white grid background for all plots
plt.rc("figure", autolayout=True)                # Ensures figure layout is automatically adjusted
plt.rc(
    "axes",
    labelweight="bold",                          # Makes the axis labels bold
    labelsize="large",                           # Sets the size of the axis labels to large
    titleweight="bold",                          # Makes the axis titles bold.
    titlesize=14,                                # Sets the size of the axis titles to 14
    titlepad=10,                                 # Adds padding of 10 points between the axis title and the plot
)

# Warnings Configuration
warnings.filterwarnings('ignore')                # Ignore all warning messages during the execution

# Directory Configuration
current_path = os.getcwd()                       # Retrieves the path of the current working directory and stores it in the current_path variable
dataset_dir = Path(current_path + "/dataset/")   # Directory for dataset files stored
input_dir = Path(current_path + "/input/")       # Directory for input files or data stored
output_dir = Path(current_path + "/output/")     # Directory for output files (e.g., processed data, results) will be saved
model_dir = Path(current_path + "/models/")      # Directory for machine learning models might be saved or loaded from
image_dir = Path(current_path + "/images/")      # Directory for plots or visualizations will be stored

# Print Paths
print('current_path:', current_path)
print('dataset_dir:', dataset_dir)
print('input_dir:', input_dir)
print('output_dir:', output_dir)
print('model_dir:', model_dir)
print('image_dir:', image_dir)
```

Figure 5.3: Configuration

split image

```
current_path: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml
dataset_dir: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/dataset
input_dir: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/input
output_dir: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/output
model_dir: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models
image_dir: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/images
```

Figure 5.4: Configuration - Print Paths

- **Matplotlib Defaults**

  Configuring Matplotlib Style (`plt.style.use`): This command specifies the global style for Matplotlib plots by utilising Seaborn's `whitegrid` theme, resulting in plots with a white background and grid lines. The reference to `v0_8` denotes a specific version or configuration. Adjusting Figure Auto Layout (`plt.rc("figure", autolayout=True)`) ensures that the figure layout is automatically modified to prevent overlapping content such as axis labels and titles. Customizing Axes Configuration (`plt.rc`) The remaining `plt.rc`

commands customise the appearance of axes in the plots. `labelweight= "bold"` enhances the visibility of axis labels by making them bold. `labelsize= "large"` sets the size of the axis labels to be large. `titleweight="bold"` bolden the axis titles. `titlesize=14` sets the size of the axis titles to 14 points. `titlepad=10` adds 10 points of padding between the axis title and the plot, enhancing readability.

- **Warnings Configuration**

  Suppressing Warnings (`warnings.filterwarnings('ignore')`) This code line sets up Python to suppress all warning messages while the code is running. This can help produce a neater output, mainly when anticipated non-critical warnings are not a concern.

- **Directory Configuration**

  The current working directory's Path can be obtained using `os.getcwd()`. This Path is where the script is executed and stored in the variable `current_path`. The paths for different directories relative to the current working directory are defined in the following lines. `dataset_dir` Path to the `dataset` directory, where the dataset files are typically stored. `input_dir` Path to the `input` directory may contain input files or data required for the script. `output_dir` Path to the `output` directory, where output files such as processed data and results will be saved. `model_dir` Path to the `models` directory, where machine learning models may be saved or loaded. `image_dir` Path to the `images` directory, where plots or visualisations will be stored.

- **Print Paths**

  Printing Paths to Directories displays the paths for the current working directory and the specified subdirectories. This is useful for confirming the correctness of the paths and serves as a convenient reference during development and debugging.

5.4 Generate and Save Dataset to CSV file

Creating a synthetic dataset to train and test an ideal egg-hatch environment involves using Python to generate data that mimics the conditions present during the setter and hatcher phases, considering temperature and humidity parameters. Throughout the initial 18 days of the setter phase, it's advised to maintain temperature levels for setters and hatchers (incubators set temperature) between `37.2` and `38.2°C`. As for the final three days of the hatcher phase, the temperature should be adjusted to fall between `36.0` and `36.5°C`. It is recommended to uphold a relative humidity (RH) of `55-65%` during the first 18 days and elevate it to around `70%` after the eggs have been moved to the hatcher.

```python
# Set random seed for reproducibility
np.random.seed(42)

# Number of samples
num_samples_ideal_setter = 400  # Ideal samples for the setter phase (first 18 days)
num_samples_ideal_hatcher = 80  # Ideal samples for the hatcher phase (last 3 days)
num_samples_non_ideal_setter = 100  # Non-ideal samples for the setter phase
num_samples_non_ideal_hatcher = 20  # Non-ideal samples for the hatcher phase

# Temperature and humidity ranges
setter_temp_range = (37.2, 38.2)
hatcher_temp_range = (36.0, 36.5)
setter_humidity_range = (55, 65)
hatcher_humidity_range = (70, 75)

# Generate synthetic data for the ideal setter phase
setter_temps_ideal = np.random.uniform(low=setter_temp_range[0], high=setter_temp_range[1], size=num_samples_ideal_setter)
setter_humidities_ideal = np.random.uniform(low=setter_humidity_range[0], high=setter_humidity_range[1], size=num_samples_ideal_setter)
setter_condition_labels_ideal = ['Ideal'] * num_samples_ideal_setter
```

Figure 5.5: Generate and Save Dataset to CSV file

The code creates artificial data for two stages of an incubation process: the setter phase (first 18 days) and the hatcher phase (last three days). It establishes a random seed to ensure reproducibility and specifies sample sizes for optimal and suboptimal conditions in both stages. The code helps generate synthetic datasets that mimic various incubation conditions, allowing for additional analysis or model training.

- **Generate Data**

  For the ideal setter phase, 400 samples were produced, with temperatures ranging from `37.2°C` to `38.2°C` and humidities between `55%` and `65%`. As for the ideal hatching phase, `80` samples were generated with temperatures between `36.0°C` and `36.5°C` and humidities between `70%` and `75%`. In the case of the non-ideal setter phase, 100 samples were created with slightly lower temperatures (ranging from `36.0°C` to `37.0°C`) and humidities between `50%` and `54%`. Lastly, for the non-ideal hatcher phase, `20` samples were

produced with slightly lower temperatures (ranging from `35.0°C` to `35.9°C`) and humidities between `65%` and `69%`.

- **Combine Data**

  Combines ideal and non-ideal samples for both phases into single arrays. Creates Pandas DataFrames for the setter and hatcher phases, with columns for temperature, humidity, and condition labels.

- **Display Data**

  Displays the initial rows of every DataFrame to offer a quick look at the data that has been created.

Below is a sample of the results of the datasets generated for Setter and Hatcher Data.

Table 5.1: Sample of Generated Setter Data (truncated) (500 rows, 3 columns)

	Temperature (°C)	Humidity (%)	Condition
0	37.574540	56.031239	Ideal
1	38.150714	64.025529	Ideal
2	37.931994	60.052524	Ideal
3	37.798658	63.264575	Ideal
4	37.356019	58.200496	Ideal
..	...	...	...
495	36.350712	53.010867	Non-Ideal
496	36.767188	50.634420	Non-Ideal
497	36.401931	53.523483	Non-Ideal
498	36.479876	53.487374	Non-Ideal
499	36.627505	50.116989	Non-Ideal

5.5 Split Data

The following instructions outline the data preparation process in the setter phase of an incubation process. The code loads the setter phase dataset from a CSV

Table 5.2: Sample of Generated Hatcher Data (truncated) (100 rows, 3 columns)

	Temperature (°C)	Humidity (%)	Condition
0	36.353619	71.155374	Ideal
1	36.076270	73.359464	Ideal
2	36.288144	70.098553	Ideal
3	36.303358	70.520543	Ideal
4	36.212065	73.999580	Ideal
..	...	...	...
95	35.485987	68.486147	Non-Ideal
96	35.701861	68.893956	Non-Ideal
97	35.096283	68.875511	Non-Ideal
98	35.684925	67.998607	Non-Ideal
99	35.487140	65.520345	Non-Ideal

file called `setter_data.csv`. Next, it extracts the features (`Temperature` and `Humidity`) into `X_setter` and the labels (`Condition`) into `y_setter`. Afterwards, the dataset is split into training and test sets using a 70-30 split, with a random seed (`random_state=42`) for reproducibility. Finally, the code prints the shapes of the training and test datasets to verify the split. This code forms a crucial initial step in preparing the data for training machine learning models for the setter phase.

```
# ---------------------------------- Setter Phase ----------------------------------
# Load the datasets
setter_data = pd.read_csv(dataset_dir / 'setter_data.csv')

# Prepare features and labels for setter phase
X_setter = setter_data[['Temperature', 'Humidity']]
y_setter = setter_data['Condition']

# Split the setter data into training and test sets
X_setter_train, X_setter_test, y_setter_train, y_setter_test = train_test_split(
    X_setter, y_setter, test_size=0.3, random_state=42
)

# Check the dataset
print(f'Setter Train data shape: X{X_setter_train.shape}, y{y_setter_train.shape}')
print(f'Setter Test data shape: X{X_setter_test.shape}, y{y_setter_test.shape}')
```

Figure 5.6: Split Data

5.6 Train Models

In this section, we explore the process of developing machine-learning models to optimise the egg-hatching environment during the setter phase. This section is divided into two parts. The first part is Logistic Regression, where we train a Logistic Regression model to classify conditions during the setter phase. The second part is SVM, where we train a Support Vector Machine (SVM) Model with a linear kernel to handle the same task. This is particularly effective for linearly separable data. Based on sensor data, these models aim to predict and maintain optimal hatching conditions.

5.6.1 Logistic Regression

Figure 5.7: Logistic Regression - Initializes and Train Model

The code sets up and trains a Logistic Regression model for the setter phase of an incubation process. This code is vital in creating a predictive model that can categorise the conditions during the setter phase as ideal or non-ideal, depending on the input features. Here's an explanation.

- **Initialize the Logistic Regression Model**

  `logistic_model_setter = LogisticRegression(max_iter=1000)` A Logistic Regression model is established with a limit of 1,000 iterations for the optimisation algorithm to reach convergence. This setting (`max_iter=1000`) guarantees that the Model has ample iterations to discover the best solution, particularly when working with potentially intricate datasets.

- **Train the Logistic Regression Model**

  `logistic_model_setter.fit(X_setter_train, y_setter_train)` The model is trained by using the training data (`X_setter_train` for features and

`y_setter_train` for labels). The `.fit()` method fine-tune the model parameters to match the training data best, identifying the optimal weights that minimise the error when predicting the `Condition` (Ideal or Non-Ideal) based on the `Temperature` and `Humidity`.

5.6.2 SVM

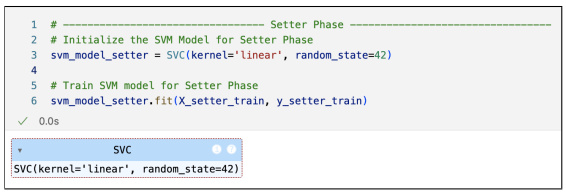


Figure 5.8: SVM - Initializes and Train Model

The code initialises and trains a Support Vector Machine (SVM) model with a linear kernel, suitable for linearly separable data in the setter phase of the egg-hatching process. It creates an SVM model and then trains it using the training data from the setter phase. During training, the Model learns to classify conditions as "Ideal" or "Non-Ideal" based on features such as temperature and humidity. Once trained, this Model can predict whether new, unseen data from the setter phase corresponds to "Ideal" or "Non-Ideal" conditions. Each part of the code serves the following purposes.

- **Initialize the SVM Model**

  The code `SVC(kernel='linear', random_state=42)` initialises a Support Vector Machine (SVM) classifier with a linear kernel. By setting `kernel='linear'`, we specify that the SVM will utilise a linear kernel function. The linear kernel is typically employed when the data is linearly separable, indicating it can be separated by a straight line (or hyperplane in higher dimensions). Additionally, by including `random_state=42`, we set the random seed to 42, ensuring reproducible results. This consistency ensures that random processes within the model training, such as data shuffling, remain consistent across different runs.

- **Train the SVM Model**

  The code `svm_model_setter.fit(X_setter_train, y_setter_train)` is used to train the SVM model with the training data for the setter phase. `X_setter_train` represents the features (temperature and humidity) from the training data of the setter phase. Meanwhile, `y_setter_train` corresponds to the labels (either "Ideal" or "Non-Ideal") for the training data of the setter phase.

5.7 Save and Load Models

The Save and Load Models section explains the essential steps for saving and reusing trained machine learning models. It details how to save Logistic Regression and SVM models using unique file names after training. Additionally, it describes loading these saved models from the disk, making it possible to reuse the trained models for future predictions and analysis efficiently.

5.7.1 Logistic Regression

Save Model

The code is created to store trained Logistic Regression models for the "setter" and "hatcher" stages in uniquely named files. Here's a breakdown of each section.

```python
# Get the current date and time
now = datetime.now()

# Format the date and time as yyyymmddtime
formatted_date_time = now.strftime("%Y%m%d%H%M%S")

# Count the number of train rows for setter and hathcer
row_count_setter_train = len(y_setter_train)
row_count_hatcher_train = len(y_hatcher_train)

# define filename for Model files
filenames_logistic_model_setter = model_dir / (f'logistic_model_setter_{formatted_date_time}_{row_count_setter_train}.pkl')
filenames_logistic_model_hatcher = model_dir / (f'logistic_model_hatcher_{formatted_date_time}_{row_count_hatcher_train}.pkl')

try:
    # Save the trained Logistic Regression model to a file
    joblib.dump(logistic_model_setter, filenames_logistic_model_setter)
    joblib.dump(logistic_model_hatcher, filenames_logistic_model_hatcher)
    print(f"Model was successfully saved!")
    print(f"logistic_model_setter: {filenames_logistic_model_setter}")
    print(f"logistic_model_hatcher: {filenames_logistic_model_hatcher}")

except Exception as e:
    print(f"Failed to save models: {e}")
```

Figure 5.9: Logistic Regression - Save Model

```
Model was successfully saved!
logistic_model_setter: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/logistic_model_setter_20240824205841_350.pkl
logistic_model_hatcher: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/logistic_model_hatcher_20240824205841_70.pkl
```

Figure 5.10: Logistic Regression - Save Model Output

- **Getting the Current Date and Time**

  Retrieve the date and time when the code is executed with this line.

- **Formatting the Date and Time**

  The code `formatted_date_time = now.strftime("%Y%m%d%H%M%S")` is used to convert the current date and time into a specific string format: "yyyymmddHHMMSS". An example of this format is "20240825153045", corresponding to "August 25, 2024, at 15:30:45". This formatted string is essential for uniquely identifying the creation time of files in their filenames.

- **Counting the Number of Training Rows**

  The following code computes the total number of rows in the training datasets for the "setter" and "hatcher" stages. The filenames include the row counts to offer more context about the data used to train the models.

- **Defining Filenames for Model Files**

  These statements establish the file names for the saved model files by including the current date and time (`formatted_date_time`) and the number of rows used for training
  (`row_count_setter_train` and `row_count_hatcher_train`). The files will be stored in the directory specified by `model_dir`, with names such as `logistic_model_setter_20230825143059_150.pkl`.

- **Saving the Models**

  This section aims to store the
  `logistic_model_setter` and `logistic_model_hatcher` models in the specified filenames using `joblib.dump`. If the storage process is successful, a confirmation message is displayed, indicating the complete paths of the stored files. If an error occurs during the storage, it is captured by the `except` section, and an error message is displayed.

Load Model

The following code is created to load Logistic Regression models previously saved in specific files on the disk. The code aims to load pre-trained Logistic Regression models from the designated file paths. If the load is successful, the models will be stored in the variables `logistic_model_setter` and `logistic_model_hatcher`. The code explains whether the model load process was successful, which can help identify potential issues. Here's an explanation of the functionality of each part of the code.

```
# set filename for Model files
filenames_logistic_model_setter = model_dir / 'logistic_model_setter_20240824205841_350.pkl'
filenames_logistic_model_hatcher = model_dir / 'logistic_model_hatcher_20240824205841_70.pkl'

try:
    # Load the saved Logistic Regression model from the file
    logistic_model_setter = joblib.load(filenames_logistic_model_setter)
    logistic_model_hatcher = joblib.load(filenames_logistic_model_hatcher)
    print("Model loaded successfully")
    print(f"logistic_model_setter: {filenames_logistic_model_setter}")
    print(f"logistic_model_hatcher: {filenames_logistic_model_hatcher}")
except Exception as e:
    print(f"Failed to load the models: {e}")
```

Figure 5.11: Logistic Regression - Load Model

```
Model loaded successfully
logistic_model_setter: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/logistic_model_setter_20240824205841_350.pkl
logistic_model_hatcher: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/logistic_model_hatcher_20240824205841_70.pkl
```

Figure 5.12: Logistic Regression - Load Model Output

- **Set Filename for Model Files**

  These lines specify the Logistic Regression models' file paths saved earlier. The filenames include details such as the date and time the models were saved (`20240824205841`) and the number of training samples used (`350` for the setter model and `70` for the hatcher model). `model_dir` is the directory where the models are stored, and the paths are constructed by appending the filenames to this directory.

- **Loading the Saved Logistic Regression Models**

  In the `try` block, the `joblib.load` function is utilised to load the models from the specified files. Upon successfully loading the models, a confirmation message, along with the paths of the loaded files, is printed. In the `except` block, if an error occurs during the model loading process (for example, if

the file does not exist or encounters a problem), the exception is captured, and an error message indicating the inability to load the models is printed.

5.7.2 SVM

Save Model

This particular piece of code is designed to save trained Support Vector Machine (SVM) models to files for future use. The code stores trained SVM models for the setter and hatcher phases into individually named files containing a timestamp and the number of training rows. A confirmation message is displayed when the saving process is successful, thanks to the `try` block. In case of an issue, the `except` block captures the error and prints a relevant message. Here's a comprehensive breakdown of the functionality of each section of the code.

```python
# Get the current date and time
now = datetime.now()

# Format the date and time as yyyymmddtime
formatted_date_time = now.strftime("%Y%m%d%H%M%S")

# Count the number of train rows for setter and hathcer
row_count_setter_train = len(y_setter_train)
row_count_hatcher_train = len(y_hatcher_train)

# define filename for Model files
filenames_svm_model_setter = model_dir / (f'svm_model_setter_{formatted_date_time}_{row_count_setter_train}.pkl')
filenames_svm_model_hatcher = model_dir / (f'svm_model_hatcher_{formatted_date_time}_{row_count_hatcher_train}.pkl')

try:
    # Save the trained SVM model to a file
    joblib.dump(svm_model_setter, filenames_svm_model_setter)
    joblib.dump(svm_model_hatcher, filenames_svm_model_hatcher)
    print(f"Model was successfully saved!")
    print(f"svm_model_setter: {filenames_svm_model_setter}")
    print(f"svm_model_hatcher: {filenames_svm_model_hatcher}")

except Exception as e:
    print(f"Failed to save models: {e}")
```

Figure 5.13: SVM - Save Model

```
Model was successfully saved!
svm_model_setter: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/svm_model_setter_20240824211525_350.pkl
svm_model_hatcher: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/svm_model_hatcher_20240824211525_70.pkl
```

Figure 5.14: SVM - Save Model Output

- **Get the Current Date and Time**

  Retrieve the date and time when the code is executed with this line.

- **Format the Date and Time**

  The code `formatted_date_time = now.strftime("%Y%m%d%H%M%S")` is used to convert the current date and time into a specific string format: "yyyymmd-

dHHMMSS". An example of this format is "20240825153045", corresponding to "August 25, 2024, at 15:30:45". This formatted string is essential for uniquely identifying the creation time of files in their filenames.

- **Counting the Number of Training Rows**

  The following code computes the total number of rows in the training datasets for the "setter" and "hatcher" stages. The filenames include the row counts to offer more context about the data used to train the models.

- **Defining Filenames for Model Files**

  The code creates file names for the saved SVM model files by utilising the directory `model_dir` and including the formatted date/time and row counts in the file names. The file names will be in the format `svm_model_setter_yyyymmddHHMMSS_rowcount.pkl` for the setter model and a comparable format for the hatcher model.

- **Try Block: Saving the Trained SVM Models**

  The `joblib.dump` function saves the trained SVM models (`svm_model_setter` and `svm_model_hatcher`) to the designated files. A confirmation message and the paths to the saved files are displayed upon successfully saving the models.

- **Except Block: Handling Errors**

  During the saving process, if an error occurs (such as problems with file paths or permissions), the program catches the exception and prints an error message to indicate the failure.

Load Model

The following code loads Support Vector Machine (SVM) models saved in files. Below is a comprehensive description of the code.

- **Set Filename for Model Files**

  The following lines designate the file names for the SVM models that were previously saved. These file names contain a timestamp (`20240824211525`)

```python
# set filename for Model files
filenames_svm_model_setter = model_dir / 'svm_model_setter_20240824211525_350.pkl'
filenames_svm_model_hatcher = model_dir / 'svm_model_hatcher_20240824211525_70.pkl'

try:
    # Load the saved SVM model from the file
    svm_model_setter = joblib.load(filenames_svm_model_setter)
    svm_model_hatcher = joblib.load(filenames_svm_model_hatcher)
    print("Model loaded successfully")
    print(f"svm_model_setter: {filenames_svm_model_setter}")
    print(f"svm_model_hatcher: {filenames_svm_model_hatcher}")
except Exception as e:
    print(f"Failed to load the models: {e}")
```

Figure 5.15: SVM - Load Model

```
Model loaded successfully
svm_model_setter: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/svm_model_setter_20240824211525_350.pkl
svm_model_hatcher: /Users/nofrialhadi/nordic/thingy53/egg_hatch/ml/models/svm_model_hatcher_20240824211525_70.pkl
```

Figure 5.16: SVM - Load Model Output

and the number of rows utilised for training (`350` for the setter model and `70` for the hatcher model). `model_dir` represents the directory path where the models are stored. The file names are generated by concatenating the specific file names to `model_dir` using the `/` operator, a functionality of the `Pathlib` library in Python that aids in managing file paths.

- **Try Block: Loading the Saved SVM Models**

  The code within the `try` block tries to load the SVM models from the specified files using the `joblib.load` function. We have two variables, `svm_model_setter` and `svm_model_hatcher`, which will store the loaded models. If the models are loaded successfully, the code will print a confirmation message and show the paths of the loaded model files.

- **Except Block: Handling Errors**

  If loading the models is problematic (such as missing, corrupted, or inaccessible files), the code will detect the issue and display an error message. The error message will contain information about the exception (`e`), which can be helpful for troubleshooting.

The following code is designed to reload SVM models that have been previously trained from files into the program. This functionality is valuable for utilising the models on new data without retraining them. A message is displayed if the models are loaded successfully due to the `try` block. If there is an error during loading, the `except` block captures the error and outputs a message indicating the failure.

5.8 Predict Label

This section explains how trained machine-learning models are used to classify sensor data. It discusses logistic regression and SVM models and shows how they can be used to predict conditions during the setter phase of the egg incubation process. The section also evaluates the Model's performance on training and test data, providing insights into its predictive accuracy and effectiveness in real-world applications.

5.8.1 Logistic Regression

```
# ---------------------------------- Setter Phase ----------------------------------
# Predict labels for the setter test data
y_setter_pred = logistic_model_setter.predict(X_setter_test)

# Calculate predictions on the training data (for Ein)
y_setter_train_pred = logistic_model_setter.predict(X_setter_train)
```

Figure 5.17: Logistic Regression - Predict Label

The trained Logistic Regression model is used to make predictions for the setter phase, and its performance on both the training and test data is evaluated. We can assess the Logistic Regression model's accuracy and other performance metrics by comparing the actual labels with the predicted labels, both in-sample (Ein) and out-of-sample (Eout). Each line of code has the following purpose.

- **Predict Labels for the Setter Test Data**

  `logistic_model_setter` predicts on the test data (`X_setter_test`) with `predict(X_setter_test)`. The predicted labels, "Ideal" or "Non-Ideal", are generated for each data point in the test set. These predictions are then stored in the array `y_setter_pred`.

- **Calculate Predictions on the Training Data (for Ein)**

  Using the `predict` method, the `logistic_model_setter` predicts labels for the training data `X_setter_train`. This enables the calculation of the in-sample error (Ein), which assesses the Model's performance on the training data. The array `y_setter_train_pred` stores the predicted labels for the training set.

- **Interpretation**

  `y_setter_pred` stores the predicted labels for the test dataset. This data will assess the Model's performance based on previously unseen data. `y_setter_train_pred` contains the predicted labels for the training dataset and will be used to evaluate the Model's performance on the data it was trained on.

5.8.2 SVM

```
# ---------------------------------- Setter Phase ----------------------------------
# Predict for Setter Phase
y_setter_pred_train = svm_model_setter.predict(X_setter_train)
y_setter_pred_test = svm_model_setter.predict(X_setter_test)
✓ 0.0s

# ---------------------------------- Hatcher Phase ----------------------------------
# Predict for Hatcher Phase
y_hatcher_pred_train = svm_model_hatcher.predict(X_hatcher_train)
y_hatcher_pred_test = svm_model_hatcher.predict(X_hatcher_test)
```

Figure 5.18: SVM - Predict Label

The code makes predictions using the trained SVM model for the setter stage of the egg incubation procedure. Here's an overview of each segment of the code's functionality.

- **Predict on Training Data**

  `y_setter_pred_train` will hold the predicted labels for the training data (`X_setter_train`) using the trained SVM model (`svm_model_setter`). The trained SVM model is used with the `svm_model_setter.predict(X_setter_train)` method to predict the labels ("Ideal" or "Non-Ideal") for the training features (`X_setter_train`), and the predicted labels are then saved in `y_setter_pred_train`.

- **Predict on Test Data**

  `y_setter_pred_test` will hold the predicted labels for test data (`X_setter_test`) by using the trained SVM model. The method `svm_model_setter.predict(X_setter_test)` will predict the labels for the test features (`X_setter_test`) using the trained SVM model and save the predicted labels in `y_setter_pred_test`.

- **Interpretation**

  The SVM model that was trained earlier is utilised to make predictions for both the training and test sets. `y_setter_pred_train` contains the predicted labels for the training data, enabling the evaluation of the Model's learning of the training data (e.g., computation of in-sample error, `Ein`). `y_setter_pred_test` holds the predicted labels for the test data, enabling the assessment of the Model's performance on unseen data (e.g., computation of out-of-sample error, `Eout`).

5.9 Evaluate Model

The Evaluate Model section assesses the trained machine learning models' performance, focusing on Logistic Regression and SVM models. Specifically, it analyses their effectiveness in predicting optimal conditions during egg hatching. This section provides a detailed analysis of each Model's predictive accuracy and reliability, offering valuable insights into their performance.

5.9.1 Logistic Regression

```python
# ---------------------------------- Setter Phase ----------------------------------
# Evaluate the model for setter phase
accuracy_setter = accuracy_score(y_setter_test, y_setter_pred)
precision_setter = precision_score(y_setter_test, y_setter_pred, pos_label='Ideal')
recall_setter = recall_score(y_setter_test, y_setter_pred, pos_label='Ideal')
f1_setter = f1_score(y_setter_test, y_setter_pred, pos_label='Ideal')
conf_matrix_setter = confusion_matrix(y_setter_test, y_setter_pred)
report_setter = classification_report(y_setter_test, y_setter_pred)

print("Setter Phase Model Evaluation")
print(f'Accuracy: {accuracy_setter:.2f}')
print(f'Precision: {precision_setter:.2f}')
print(f'Recall: {recall_setter:.2f}')
print(f'F1 Score: {f1_setter:.2f}')
print('Confusion Matrix:')
print(conf_matrix_setter)
print('Classification Report:')
print(report_setter)

# Calculate Ein for setter phase
accuracy_setter_train = accuracy_score(y_setter_train, y_setter_train_pred)
Ein_setter = 1 - accuracy_setter_train
```

Figure 5.19: Logistic Regression - Evaluate Model

The following code assesses the Logistic Regression model's performance on the setter phase data. It delivers an in-depth analysis of its accuracy, precision, recall, F1 score, confusion matrix, and error rates. Here's a detailed overview of each segment.

```
Setter Phase Model Evaluation
Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1 Score: 1.00
Confusion Matrix:
[[119   0]
 [  0  31]]
Classification Report:
              precision    recall  f1-score   support

       Ideal       1.00      1.00      1.00       119
   Non-Ideal       1.00      1.00      1.00        31

    accuracy                           1.00       150
   macro avg       1.00      1.00      1.00       150
weighted avg       1.00      1.00      1.00       150

Setter Phase Errors
Ein (In-sample error): 0.00
Eout (Out-of-sample error): 0.00
Difference: 0.00
```

Figure 5.20: Logistic Regression - Evaluate Model Output

- **Model Evaluation**

  Accuracy:

  `accuracy_setter = accuracy_score(y_setter_test, y_setter_pred)` The Model's accuracy on the test data is computed as the percentage of correct predictions compared to the total number of predictions.

  Precision:

  `precision_setter` is assigned the value of `precision_score(y_setter_test, y_setter_pred, pos_label='Ideal')`. Precision assesses the ratio of true positives (accurate "Ideal" predictions) to all positive predictions (including both true and false positives). This metric evaluates the precision of the Model's positive predictions.

  Recall:

  `recall_setter = recall_score(y_setter_test, y_setter_pred, pos_label='Ideal')` The recall score, also called sensitivity, calculates the percentage of true positives the Model correctly identified. It assesses the Model's capability to detect all positive instances.

  F1 Score:

  `f1_setter = f1_score(y_setter_test, y_setter_pred, pos_label='Ideal')`

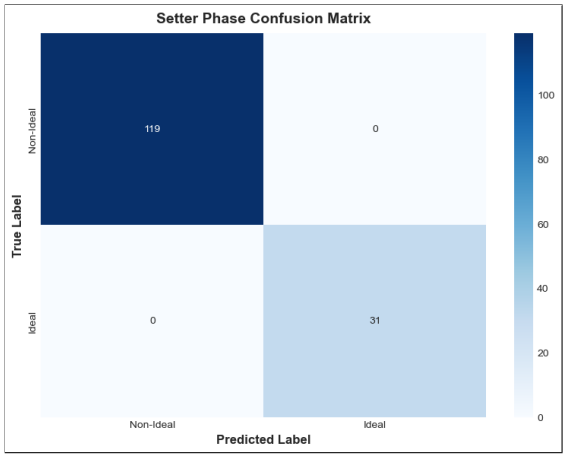


Figure 5.21: Logistic Regression - Setter Phase Confusion Matrix

The F1 score represents the harmonic mean of precision and recall, giving a unified measure that considers both aspects. This is especially beneficial when dealing with imbalanced class distributions.

Confusion Matrix:

`conf_matrix_setter = confusion_matrix(y_setter_test, y_setter_pred)`

The confusion matrix is a concise representation of the prediction outcomes, displaying the number of true positives, true negatives, false positives, and false negatives. It offers a comprehensive insight into the Model's accuracy for each category.

Classification Report:

`report_setter = classification_report(y_setter_test, y_setter_pred)`

The classification report provides a comprehensive performance overview, encompassing precision, recall, F1 score, and support for every class.

Printing Results:

The code displays the accuracy, precision, recall, F1 score, confusion matrix, and classification report, providing insight into the Model's performance on the test data.

- **Error Calculation**

  Ein (In-sample error):

The training accuracy, represented by `accuracy_setter_train`, is calculated using the `accuracy_score` function with `y_setter_train` and `y_setter_train_pred` as parameters. The in-sample error, denoted as `Ein_setter`, is computed by subtracting the training accuracy from 1. This metric assesses the Model's performance on the training data.

Eout (Out-of-sample error):
`Eout_setter = 1 - accuracy_setter` To calculate the out-of-sample error (Eout), subtract the test accuracy from 1. This evaluates the Model's ability to generalise to unseen data.

Difference between Ein and Eout:
`Diff_setter` computes the absolute value of the difference between `Ein_setter` and `Eout_setter`. This calculation is used to evaluate the Model's overfitting. If there is a significant difference, it indicates overfitting, which happens when the Model performs well on the training data but poorly on the test data.

Printing Error Results:
The code displays Ein, Eout, and the variance between them to help understand the Model's generalisation abilities.

- **Heatmap of Confusion Matrix**

  The code produces a heatmap to display the confusion matrix using `sns.heatmap`. The visual depiction aids in the straightforward interpretation of both correct and incorrect predictions for each category ("Ideal" and "Non-Ideal").

The Logistic Regression model is extensively assessed using the setter phase data, thoroughly evaluating its in-sample and out-of-sample performance. The assessment includes a detailed examination of the Model's accuracy, precision, recall, F1 score, and the possibility of overfitting by comparing Ein and Eout. A heatmap of the confusion matrix provides a visual representation of the Model's classification performance.

The code's output evaluates the performance of the Logistic Regression model on the test data during the setter phase. Below is an analysis of the results.

- **Model Evaluation Output**

  Accuracy: 1.00
  The Model obtained a 100% accuracy, accurately categorising all samples in the test set.

  Precision: 1.00
  The precision for both classes ("Ideal" and "Non-Ideal") is 100%, suggesting that the Model's predictions for "Ideal" and "Non-Ideal" instances were accurate.

  Recall: 1.00
  The recall for both categories is also 100%. This indicates that the Model correctly recognised all genuine "Ideal" and "Non-Ideal" samples.

  F1 Score: 1.00
  The F1 score, which equally weighs precision and recall, achieves a perfect score of 1.00 for both classes. This suggests that the Model correctly identifies positive cases and reduces false positives.

  Confusion Matrix:
  The confusion matrix indicates that 119 samples labelled as "Ideal" were accurately classified (true positives). 31 samples labelled "Non-Ideal" were accurately classified (true negatives). There were no instances of false positives or false negatives, demonstrating impeccable classification.

  Classification Report:
  The report on classification verifies the precision, recall, and F1 scores for each category, all of which are at 1.00, indicating flawless model performance.

- **Error Calculation Output**

  Ein (In-sample error): 0.00
  The Model's in-sample error is 0.00, indicating that it made no errors in the training data.

  Eout (Out-of-sample error): 0.00
  The out-of-sample error is also 0.00, which shows that the Model did not make any errors in the test data.

Difference: 0.00

The Model's performance is steady on training and test data, with Ein and Eout showing no difference at 0.00, indicating no overfitting.

The Model performed flawlessly on the training and test data during the setter phase. This outcome shows that the Model differentiates between "Ideal" and "Non-Ideal" conditions using the temperature and humidity features. The Model shows no errors (Ein and Eout both being 0), and the confusion matrix is perfect, indicating that the Model has been generalised effectively and does not overfit the training data.

5.9.2 SVM

```
# ---------------------------------- Setter Phase ----------------------------------
# Evaluate Setter Phase
accuracy_setter_svm = accuracy_score(y_setter_test, y_setter_pred_test)
precision_setter_svm = precision_score(y_setter_test, y_setter_pred_test, pos_label='Ideal')
recall_setter_svm = recall_score(y_setter_test, y_setter_pred_test, pos_label='Ideal')
f1_setter_svm = f1_score(y_setter_test, y_setter_pred_test, pos_label='Ideal')
conf_matrix_setter_svm = confusion_matrix(y_setter_test, y_setter_pred_test)
report_setter_svm = classification_report(y_setter_test, y_setter_pred_test)

print("Setter Phase Model Evaluation")
print(f'Accuracy: {accuracy_setter_svm:.2f}')
print(f'Precision: {precision_setter_svm:.2f}')
print(f'Recall: {recall_setter_svm:.2f}')
print(f'F1 Score: {f1_setter_svm:.2f}')
print("Confusion Matrix (Setter):")
print(conf_matrix_setter_svm)
print('Classification Report:')
print(report_setter_svm)

# In-sample error (Ein)
accuracy_setter_train_svm = accuracy_score(y_setter_train, y_setter_pred_train)
ein_setter_svm = 1 - accuracy_setter_train_svm
```

Figure 5.22: SVM - Evaluate Model

The code assesses how well the SVM model performs during the setter phase of the egg-hatching process. Here's an in-depth explanation of the evaluation.

- **Model Evaluation**

  Accuracy:

  To determine the SVM model's accuracy on the test data, we must compare `y_setter_test` with `y_setter_pred_test`. This will give us the percentage of correct predictions.

  Precision:

  Precision measures the accuracy of instances predicted as "Ideal" and their actual classification as "Ideal."

```
Setter Phase Model Evaluation
Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1 Score: 1.00
Confusion Matrix (Setter):
[[119   0]
 [  0  31]]
Classification Report:
              precision    recall  f1-score   support

       Ideal       1.00      1.00      1.00       119
   Non-Ideal       1.00      1.00      1.00        31

    accuracy                           1.00       150
   macro avg       1.00      1.00      1.00       150
weighted avg       1.00      1.00      1.00       150

Setter Phase Errors
Ein (In-sample error): 0.00
Eout (Out-of-sample error): 0.00
Difference: 0.00
```

Figure 5.23: SVM - Evaluate Model Output

Recall:

Recall Measures how many actual "Ideal" instances were correctly predicted.

F1 Score:

The F1 Score is calculated as the harmonic mean of precision and recall, which helps to find a balance between the two.

Confusion Matrix:

It generates a confusion matrix that displays the number of accurate positive predictions, accurate negative predictions, incorrect positive predictions, and incorrect negative predictions.

Classification Report:

Generates an in-depth analysis outlining precision, recall, F1 score, and support (number of true instances) for every class.

- **Error Calculation**

  In-Sample Error (Ein):

  `Ein_setter_svm` Calculates the in-sample error, which represents the error rate on the training data. This is computed as `1 - accuracy on training data`.

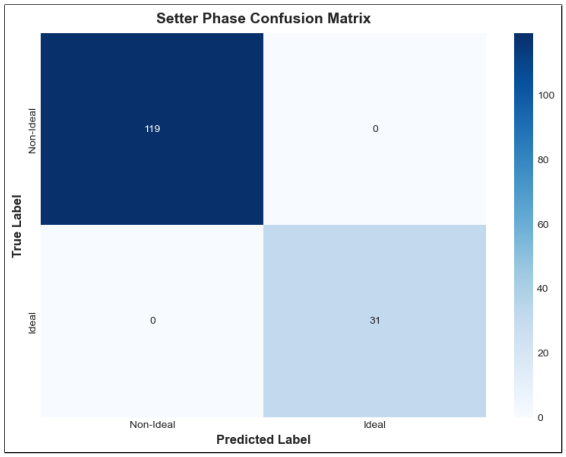


Figure 5.24: SVM - Setter Phase Confusion Matrix

Out-of-Sample Error (Eout):
The `Eout_setter_svm` function calculates the out-of-sample error, representing the error rate on the test data. This is determined by subtracting the accuracy of the test data from 1.

Difference Between Ein and Eout:
Calculating the difference between in-sample error (Ein) and out-of-sample error (Eout) can help determine whether the Model is overfitting or underfitting.

- **Visualization**

  A heatmap is created to display the confusion matrix. This helps identify the areas where the Model makes accurate and inaccurate predictions.

- **Interpretation of Output**

  The SVM model's performance on the test data will be assessed by outputting the accuracy, precision, recall, and F1 score. The confusion matrix will display the number of true positives, true negatives, false positives, and false negatives. Detailed performance metrics for each class will be provided in the classification report. The difference between the in-sample error (Ein) and out-of-sample error (Eout) will also be calculated and displayed. Lastly, a visualisation of the confusion matrix in a heatmap will be generated.

This evaluation provides a comprehensive understanding of the SVM model's performance, aiding in identifying how well it generalises to new, unseen data.

The evaluation of the SVM model's performance during the setter phase indicates it performed exceptionally well. Below is a thorough explanation.

- **Model Performance**

  Accuracy: 1.00
  The Model attained flawless precision by accurately forecasting all the labels within the test dataset.

  Precision: 1.00
  The "Ideal" class has a perfect precision. This means that all the instances the Model identified as "Ideal" were truly "Ideal."

  Recall: 1.00
  The Model's recall is flawless, indicating that it correctly recognised all instances labelled "Ideal" in the test set.

  F1 Score: 1.00
  The F1 score, which provides a trade-off between precision and recall, is ideal, indicating that the Model balances precision and recall without any trade-offs.

- **Confusion Matrix**

  The Model correctly predicted 119 instances of "Ideal", resulting in True Positives. The Model accurately predicted 31 instances of "Non-Ideal," categorised as True Negatives. There were 0 instances of "Non-Ideal" incorrectly predicted as "Ideal," categorised as False Positives. Likewise, 0 instances of "Ideal" were incorrectly predicted as "Non-Ideal," categorised as False Negatives.

- **Classification Report**

  The report on classification indicates that both "Ideal" and "Non-Ideal" classes have achieved flawless precision, recall, and F1 scores, confirming the Model's exceptional performance on the test set.

- **Error Metrics**

  Ein (In-sample error): 0.00
  The Model achieved a zero in-sample error, indicating no mistakes in the training data.

  Eout (Out-of-sample error): 0.00
  No errors were made on the test data, as the out-of-sample error is zero.

  Difference Between Ein and Eout: 0.00
  The in-sample and out-of-sample errors are the same, indicating that the Model effectively generalises to new data without overfitting.

The SVM model performs exceptionally well during the setter phase, achieving flawless accuracy, precision, recall, and F1 scores. Its flawless performance, both in-sample and out-of-sample, indicates its highly effective ability to differentiate between "Ideal" and "Non-Ideal" conditions in the setter phase while also generalising well to new data.

5.10 Predict Sensor Data Condition

In this part, we import the sensor dataset to prepare it for analysis. Then, we use our trained machine-learning models to generate predictions using the sensor inputs. The prediction results are then structured into a results dataframe, offering a well-organised and clear view of the outcomes. Lastly, this dataframe is stored in a CSV file for additional analysis and reporting.

5.10.1 Load Sensor Dataset

The following code snippet imports a Sensor dataset from a CSV file and then displays the dataset along with fundamental details about its structure. This is a typical practice in data analysis, where the dataset is imported and examined briefly before additional processing or analysis. Here's an analysis of the functionality of each section of the code.

```
# Load the datasets
sensor_data_setter_df = pd.read_csv(dataset_dir / 'sensor_data_setter.csv')

# Print the dataset
print(f"Sensor Data Setter ({sensor_data_setter_df.shape[0]} rows, {sensor_data_setter_df.shape[1]} columns):")
print(sensor_data_setter_df)
```

Figure 5.25: Load Dataset Setter

- **Loading the Sensor Dataset**

  `pd.read_csv(...)` This Pandas library function reads a CSV (Comma-Separated Values) file and imports its contents into a Pandas DataFrame. A DataFrame is a two-dimensional, labelled data structure featuring columns of different types, similar to a table in a database or an Excel spreadsheet. `dataset_dir / 'sensor_data_setter.csv'` This indicates the file path to the CSV file. The `dataset_dir` variable, defined earlier, contains the Path to the `dataset` directory. The `/` operator concatenates paths, combining `dataset_dir` with the filename `'sensor_data_setter.csv'`. `sensor_data_setter_df` The retrieved data is stored in this DataFrame variable. This variable now contains the data from the CSV file, organised with rows and columns.

- **Printing the Dataset Information**

  `sensor_data_setter_df.shape` The `shape` attribute of the DataFrame returns a tuple representing the DataFrame's dimensions.
  `sensor_data_setter_df.shape[0]` This provides the number of rows in the DataFrame. `sensor_data_setter_df.shape[1]` This provides the number of columns in the DataFrame.
  `f "Sensor Data Setter ({...} rows, {...} columns):"` This is an f-string, a way to format strings in Python by including expressions inside curly braces `{}`. In this case, it creates a message that displays the number of rows and columns in the dataset.

- **Printing the Dataset Content**

  `print(sensor_data_setter_df)` When executing this command, the entire content of the DataFrame will be displayed on the console. The output will be in the form of a table, showing all rows and columns. The display may be truncated if the DataFrame is large, showing only the first and last few rows.

The first step is to import the `sensor_data_setter.csv` file into a Pandas DataFrame named `sensor_data_setter_df`. Next, a message displays the dataset's row and column count. Lastly, the dataset's contents are printed to examine the loaded data visually. Here is a truncated Sensor Data Setter.

Table 5.3: Sensor Data Setter (truncated) (74 rows, 2 columns)

	Temperature (°C)	Humidity (%)
0	29.46	42.82
1	29.50	42.66
2	29.50	42.38
3	29.53	42.20
4	29.56	41.97
..	...	...
69	29.68	41.68
70	29.68	41.69
71	29.68	41.76
72	29.68	41.84
73	29.69	41.90

5.10.2 Prediction

The following code snippet aims to predict labels for a dataset named `sensor_data_setter_df` using two distinct machine learning models: Logistic Regression and Support Vector Machine (SVM). These predictions enable a comparison of how each Model categorises the data based on its training. The purpose of each line is explained below.

```
# Predict labels for the sensor_data_setter with Logistic Regression Model
y_sensor_data_setter_pred_log = logistic_model_setter.predict(sensor_data_setter_df)

# Predict labels for the sensor_data_setter with SVM model
y_sensor_data_setter_pred_svm = svm_model_setter.predict(sensor_data_setter_df)
```

Figure 5.26: Prediction

- **Logistic Regression Prediction**

Figure 5.27: Generate Result Dataframe

The `logistic_model_setter` is utilised to predict labels for the `sensor_data_setter_df` dataset using a trained Logistic Regression model. The predictions are saved in the variable `y_sensor_data_setter_pred_log`.

- **SVM Prediction**

  This line predicts labels for the same dataset (`sensor_data_setter_df`) using the `svm_model_setter`, a trained support vector machine (SVM) model. The variable `y_sensor_data_setter_pred_svm` holds the predictions made by the SVM model.

5.10.3 Generate Result Dataframe

The code creates two new dataframes by merging the original sensor data with the predicted labels from two different machine learning models: Logistic Regression and Support Vector Machine (SVM). The following describes the purpose of each section of the code.

- **Generate Sensor Data Output DataFrame for Logistic Regression Model**

  The original sensor data is contained in `sensor_data_setter_df`. The predicted labels from the Logistic Regression model can be found in `y_sensor_data_setter_pred_log`. To combine the original sensor data and the predicted labels along the columns, we use `pd.concat` with the parameter `axis=1`. This action creates a new dataframe named. `sensor_data_setter_output_log_df`, which now includes the original sensor data and an additional `Condition` column containing the predictions from the Logistic Regression model.

- **Generate Sensor Data Output DataFrame for SVM Model**

In the same way as previously mentioned, `sensor_data_setter_df` refers to the dataframe. `y_sensor_data_setter_pred_svm` holds the predicted labels generated by the SVM model. The original sensor data and the predicted labels from the SVM model are combined horizontally. This merging results in a new dataframe named `sensor_data_setter_output_svm_df`, which contains the sensor data and a `Condition` column representing the SVM model's predictions.

- **Display the DataFrames**

  The content of both dataframes is displayed in the console by the `print` statements. This enables us to observe both models' original sensor data and predicted labels.

The sensor data is combined with the predicted conditions from the Logistic Regression and SVM models to create two new dataframes.
`sensor_data_setter_output_log_df` includes the sensor data and predictions from the Logistic Regression model, while `sensor_data_setter_output_svm_df` includes the sensor data and predictions from the SVM model. After that, the code displays these new dataframes, allowing you to visually examine how the predictions align with the original sensor data. Below is a truncated Sensor Data Setter combined with the predicted conditions (Column "Condition") from the Logistic Regression and SVM models.

5.10.4 Save Result to CSV file

The following code stores the outcomes of the predictions produced by two machine learning models (Logistic Regression and SVM) in CSV files. Here's an explanation of the function of each section of the code.

- **Getting the Current Date and Time**

  Retrieve the current date and time with `now = datetime.now()`.

- **Formatting the Date and Time**

Table 5.4: Sensor Data Setter combined with the predicted conditions using the Logistic Regression Model (truncated) (74 rows, 3 columns)

	Temperature (°C)	Humidity (%)	Condition
0	29.46	42.82	Non-Ideal
1	29.50	42.66	Non-Ideal
2	29.50	42.38	Non-Ideal
3	29.53	42.20	Non-Ideal
4	29.56	41.97	Non-Ideal
..	...	...	...
69	29.68	41.68	Non-Ideal
70	29.68	41.69	Non-Ideal
71	29.68	41.76	Non-Ideal
72	29.68	41.84	Non-Ideal
73	29.69	41.90	Non-Ideal

The code `formatted_date_time = now.strftime("%Y%m%d%H%M%S")` is used to convert the current date and time into a specific string format: "yyyymmddHHMMSS". An example of this format is "20240825153045", corresponding to "August 25, 2024, at 15:30:45". This formatted string is essential for uniquely identifying the creation time of files in their filenames.

- **Counting the Number of Prediction Rows**

  `sensor_data_setter_df.shape[0]` calculates the total number of rows in the `sensor_data_setter_df` dataframe (which holds the sensor data). This

```
# Get the current date and time
now = datetime.now()

# Format the date and time as yyyymmddtime
formatted_date_time = now.strftime("%Y%m%d%H%M%S")

# Count the number of prediction rows
row_count_pred = sensor_data_setter_df.shape[0]

# define filename for CSV file
filenames_pred_log = output_dir / (f'sensor_data_setter_output_log_{formatted_date_time}_{row_count_pred}.csv')
filenames_pred_svm = output_dir / (f'sensor_data_setter_output_svm_{formatted_date_time}_{row_count_pred}.csv')

try:
    # Save the DataFrame to a CSV file
    sensor_data_setter_output_log_df.to_csv(
    sensor_data_setter_output_svm_df.to_csv(filenames_pred_svm, index=False)
    print(f"Prediction was successfully saved!")
    print(f"sensor_data_setter_output_log_df: {filenames_pred_log}")
    print(f"sensor_data_setter_output_svm_df: {filenames_pred_svm}")

except Exception as e:
    print(f"Failed to save predictions: {e}")
```

(variable) filenames_pred_svm: Path

Figure 5.28: Save Result to CSV file

Table 5.5: Sensor Data Setter combined with the predicted conditions using the SVM Model (truncated) (74 rows, 3 columns)

	Temperature (°C)	Humidity (%)	Condition
0	29.46	42.82	Non-Ideal
1	29.50	42.66	Non-Ideal
2	29.50	42.38	Non-Ideal
3	29.53	42.20	Non-Ideal
4	29.56	41.97	Non-Ideal
..	...	...	...
69	29.68	41.68	Non-Ideal
70	29.68	41.69	Non-Ideal
71	29.68	41.76	Non-Ideal
72	29.68	41.84	Non-Ideal
73	29.69	41.90	Non-Ideal

Figure 5.29: Save Result to CSV file - Output

value will also be part of the file names.

- **Defining Filenames for the CSV Files**

  Formatting the filenames for the CSV files that will hold the predictions involves including a specific prefix for each type of Model:
  `sensor_data_setter_output_log_` for Logistic Regression and
  `sensor_data_setter_output_svm_` for SVM. The filenames also include the formatted date and time and the number of rows in the dataset. For instance, a filename such as
  `sensor_data_setter_output_log_20240825153045_150.csv` would signify that the file was generated on August 25, 2024, at 15:30:45 and contains 150 rows of predictions.

- **Saving the DataFrames to CSV Files**

to save the two dataframes (`sensor_data_setter_output_log_df` and `sensor_data_setter_output_svm_df`) as CSV files. Adding `index=False` to ensure that the row indices are not included in the saved files, maintaining a clean output. Once the files are successfully saved, a confirmation message is printed along with the paths of the saved files.

- **Error Handling**

  Whenever the file-saving process encounters an exception, it will be caught using the `try:` and `except Exception as e:` blocks. If any issues arise, an error message will be printed to notify about the error.

6

Results and Discussion

The Results and Discussion chapter comprehensively overviews the study's findings and insights. It is divided into three main sections: System Implementation, which details the process and challenges encountered during the development of the prototype system; Data Analytics, which explores the data analysis and machine learning models used; and Discussion, where we interpret the results about the study's objectives and compare the original timelines with the actual timelines. This chapter aims to summarize the results, evaluate the effectiveness of the implemented solutions, and offer reflections on the overall success of the project and areas for improvement.

6.1 System Implementation

During the system implementation phase, we focused on setting up the platform and development environments, experimenting with the Nordic Semiconductor Thingy:53, and establishing the process for reading sensor data via API requests. The implementation aimed to build a prototype capable of obtaining sensor data from the Nordic Thingy:53, specifically using the BME688 and BH1749 sensors. The design intended to deliver this data in JSON format through an API request by establishing an HTTP server on the device. The API requests were planned to be executed from the DIME (DyWA Integrated Modeling Environment), with the sensor data displayed in DIME's user interface through graphs and diagrams. The

prototype was also designed to collect and store the sensor data in CSV format, which would later be used for machine learning and data analytics.

However, during the implementation, we encountered significant challenges. Establishing a stable USB network interface proved to be difficult, and we faced numerous build errors when attempting to set up the USB HTTP server and JSON API endpoint on the Nordic Thingy:53. Additionally, the integration of API requests with DIME was also postponed due to these technical issues. These difficulties and time constraints led us to temporarily delay the full implementation of the HTTP server, API features, and DIME integration. In the interim, we gathered sensor data directly from Thingy:53 and stored it in CSV format for subsequent use in machine learning. Despite these setbacks, the prototype solution allowed us to continue progressing with data collection and analysis, ensuring that the project's primary objectives could still be met.

6.2 Data Analytics

During the data analytics phase, we systematically constructed and evaluated machine learning models to predict optimal conditions based on collected sensor data. The workflow started with importing the dataset, which was divided into two phases: Setter and Hatcher. These phases represent critical stages in the incubation process. Splitting the data this way could better tailor our models to each stage's specific conditions and requirements.

After categorizing the data, we divided the Setter and Hatcher datasets into train and test datasets separately. This division was essential for effectively training our machine-learning models while preserving data for validation. The workflow was conducted using Python within Jupyter Notebook and integrated into the Visual Studio Code environment. This setup provided an interactive platform for both the analysis and visualization of the data, enabling us to iterate rapidly and refine our models.

We focused on two primary machine learning techniques: Logistic Regression and Support Vector Machines (SVM). These models were chosen for their ability to handle classification tasks and their robustness in dealing with varied data. We

trained the models using the prepared datasets, carefully adjusting parameters to optimize performance.

After completing the training phase, we utilized the models to predict labels for new data, which enabled us to evaluate their accuracy and generalization capabilities—the evaluation involved comparing the predicted labels with the actual conditions to measure the effectiveness of the models. In the next phase, we used the models to predict sensor data conditions and categorize them as ideal or non-ideal based on the input features. This categorization provided valuable insights for maintaining optimal conditions during the critical stages of egg incubation.

The predictions were then used to guide decision-making processes, contributing to further refinement of the incubation environment. This analytical process set the foundation for future machine learning improvements and applications in optimizing the egg incubation process.

6.3 Discussion

In the Discussion section, we will thoroughly examine the study results about the initial objectives and planned timelines. This section is organized first to evaluate how well the study achieved the objectives outlined in Chapter 1, focusing on the effectiveness and comprehensiveness of the solutions developed. Subsequently, we will compare the proposed initial timeline with the actual timeline to identify and understand any deviations that occurred. This comparison will help assess the project's progress and highlight areas where adjustments were needed. This discussion aims to provide insights into the project's success and reflect on the factors influencing its execution.

6.3.1 Addressing the Study Objectives

Assess how the study effectively tackled and achieved the objectives set out in Chapter 1. Each objective was pivotal in directing the study and development efforts, ultimately resulting in the project's successful conclusion.

- **Objective 1: Develop a prototype IoT-based system for real-time monitoring of the egg-hatching environment using the Thingy:53**

The main objective of this project was to develop an IoT-based prototype system for real-time monitoring of the egg-hatching environment. This objective was attained using the Nordic Semiconductor Thingy:53, a flexible IoT device with various sensors. The system was created to gather data from these sensors and offer instant insights into the crucial environmental conditions needed for egg incubation. Despite facing challenges in setting up a reliable USB network interface for Thingy:53 and encountering difficulties establishing the USB HTTP server and JSON API endpoint, the project successfully produced a prototype capable of gathering sensor data. This data was saved in CSV format, enabling further analysis and implementation of machine learning applications.

- **Objective 2: Incorporate various sensors of Thingy:53 (e.g., temperature, humidity, pressure, light, and air quality) to gather comprehensive environmental data**

  The full range of sensors in Thingy:53 was utilized to gather comprehensive environmental data. These sensors measure temperature, humidity, pressure, light, and air quality, all vital factors in the egg-hatching process. The information gathered from these sensors offered a detailed understanding of the conditions within the incubation environment. This data was necessary for real-time monitoring and subsequent data analysis and machine-learning tasks to improve the hatching conditions. Despite initial technical challenges, the successful integration of these sensors enabled a thorough data collection process, ensuring that all relevant environmental factors were monitored and documented.

- **Objective 3: Apply data analytics and machine learning methods to optimize hatching conditions**

  The main objective was to use data analytics and machine learning to improve the hatching conditions. This involved creating and training machine learning models, including Logistic Regression and Support Vector Machine (SVM) models. The models were trained using a dataset split into two

phases: Setter and Hatcher, which are crucial stages in the incubation process. By analyzing this data, the models could predict and classify conditions during the Setter and Hatcher phase as ideal or non-ideal based on input features. These predictions offered actionable insights to optimize the incubation environment and enhance hatchability rates. Python and Jupyter Notebook in Visual Studio Code facilitated an interactive and iterative approach to data analysis, ensuring efficient progress.

In summary, the project has successfully met all the objectives in Chapter 1. A functional prototype IoT system was developed, comprehensive environmental data was collected using the Thingy:53 sensors, and advanced data analytics and machine learning techniques were applied to optimize the egg-hatching conditions. While there were some challenges, particularly in the technical implementation, the project has laid a solid foundation for future Work.

6.3.2 Comparison Between the Original and Actual Timelines

The study was initially planned to follow a structured and sequential approach, with specific phases allocated in distinct time frames to ensure steady progress and timely completion. However, when comparing the original timeline to the actual timeline, several deviations occurred, primarily influenced by the complexities encountered during the implementation phase. Below is an explanation of the comparison between the original and actual timelines.

- **Initial Concept and Requirements Gathering**

  The Initial Concept and Requirements Gathering phase was extended by two days, ending on June 5th instead of June 3rd. This extension was necessary to thoroughly refine the project scope and ensure all requirements were fully understood before moving on to the next phase.

- **Related Work**

  The Related Work phase was extended nine days beyond the original plan, from June 18th to June 27th. This extension was necessary due to the

Table 6.1: Comparison Between the Original and Actual Timelines

Phase	Original Timeline	Actual Timeline
Initial Concept and Requirements Gathering	20/05/2024-03/06/2024	20/05/2024-05/06/2024
Related Work	04/06/2024-18/06/2024	04/06/2024-27/06/2024
System Implementation	19/06/2024-11/07/2024	24/06/2024-29/07/2024
Data Analytics	12/07/2024-26/07/2024	30/07/2024-06/08/2024
Writing and Revising	27/07/2024-17/08/2024	18/07/2024-15/08/2024
Final Revisions and Submission	18/08/2024-25/08/2024	16/08/2024-27/08/2024

date format: `"dd/mm/yyyy"`

unexpectedly large amount of literature and technologies that needed to be reviewed, especially regarding Thingy:53 and its integration possibilities with the DIME platform. This delay in completing the literature review phase had a cascading effect on the subsequent phases.

- **System Implementation**

  The System Implementation phase started later than planned and lasted longer than expected. Instead of running from June 19th to July 11th, it took place from June 24th to July 29th. The delay was due to unexpected difficulties in integrating Thingy:53 with the DIME platform and setting up the API interface. Problems with unstable USB network interfaces and build errors when configuring the HTTP server and JSON API endpoint on the Thingy:53 caused significant delays, requiring extra troubleshooting and adjusting the implementation strategy.

- **Data Analytics**

  The Data Analytics phase, initially planned from July 12th to July 26th, started on July 30th and concluded on August 6th. This delay was due to the prolonged System Implementation phase, which affected the data availability required for the analytics work. Even though this phase's duration

was shorter, all essential steps, such as model training, evaluation, and prediction, were successfully carried out. Nevertheless, the limited time for this important phase necessitated a highly focused and efficient workflow.

- **Writing and Revising**

  The Writing and Revising phase unexpectedly started on July 18th, overlapping with the Data Analytics phase, and ended on August 15th. Starting this phase early was essential to effectively manage the overall timeline, especially considering the delays in System Implementation and Data Analytics. The overlap allowed for simultaneous progress on analysis and documentation, helping to reduce some of the delays experienced in earlier phases.

- **Final Revisions and Submission**

  The Final Revisions and Submission phase began slightly earlier and ended later than originally planned, running from August 16th to August 27th. This extended period provided enough time to finalize and polish the document, ensuring that all aspects were thoroughly reviewed and refined before submission. Despite the earlier delays, this final phase closely adhered to the extended timeline, ensuring the project was completed within an acceptable timeframe.

Despite some deviations from the original timeline, particularly during the System Implementation and Data Analytics phases, these adjustments were necessary to address unforeseen challenges. The strategic overlap of the Writing and Revising phase with Data Analytics helped to offset some delays, ultimately leading to successful project completion within the adjusted timeline.

7

Conclusion and Future Work

This chapter summarises the study's key findings and reflects on its impact, significance, and limitations. The Conclusion section provides a detailed overview of the achievements, including system prototyping and data analytics, while assessing how well the study met its original objectives. The Future Work section outlines the planned steps to address the challenges encountered during the survey, mainly focusing on improving the system's network interface and enhancing its overall functionality for future applications.

7.1 Conclusion

The study's primary objective was accomplished by employing a comprehensive method that included a systems prototype, analyzing data, and evaluating the study's objectives. Nordic Thingy:53 sensors were used to test an IoT-based system, enabling real-time monitoring of the egg-hatching environment and collecting crucial environmental data such as temperature, humidity, and air quality. Throughout the data analysis phase, the data was examined using machine learning models such as Logistic Regression and Support Vector Machines (SVM), which provided predictive insights for improving incubation conditions. The assessment of the study's progress, including comparing the original and actual timelines, revealed deviations due to unforeseen challenges. Despite these challenges, the study remained on course to achieve its objectives.

The study successfully met its objectives and provided valuable insights into using IoT and machine learning for environmental monitoring. The system's capacity to collect and analyze real-time data establishes a solid foundation for optimizing incubation conditions.

Despite the study's accomplishments, several limitations were encountered. Difficulties establishing a stable USB network interface and various build errors during the setup of the USB HTTP server and JSON API endpoint on the Nordic Thingy:53 posed significant challenges. These technical issues caused delays, necessitating adjustments to the study timeline. Although some of these issues were addressed by postponing specific tasks and focusing on data collection and analysis, it is essential to resolve these limitations for future system iterations. Addressing these technical challenges is crucial for enhancing the system's reliability and functionality in subsequent development phases.

7.2 Future Work

Encountering issues when setting up the USB network interface and establishing the USB HTTP server with a JSON API endpoint on the Nordic Thingy:53 emphasized the need for a more reliable solution. The complications and instability associated with using USB as the network interface posed a significant challenge, resulting in several build errors and project timeline delays.

To address these issues, future efforts will integrate a physical network hardware device, specifically the nRF7002 Expansion Board (nRF7002 EB), to substitute the USB network interface. This expansion board provides low-energy Wi-Fi 6 capabilities, which is expected to deliver a more secure and stable network connection for the IoT-based system. Using the nRF7002 EB, we aim to establish a more dependable HTTP server and JSON API endpoint, enabling smooth access to and control of the sensor data collected by Thingy:53.

Upon acquiring the nRF7002 EB, we will continue our exploration with this improved configuration, tackling the previous technical constraints and enhancing the system's overall performance. This hardware upgrade is expected to resolve the issues encountered with the USB interface and improve the system's efficiency

and reliability in real-time monitoring of the egg-hatching environment. Additionally, this future work will investigate further optimization of data transmission and processing capabilities, ensuring that the system can fully capitalize on the enhanced network infrastructure to support advanced machine learning and data analytics tasks.

Bibliography

[1] N. Thingy:53, "Nordic thingy:53 - nordicsemi.com," 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-53 xiii, 4, 6, 13, 14, 15, 16, 34

[2] DIME, "Introduction dime wiki," 2021. [Online]. Available: https://scce.gitlab.io/dime/content/introduction/ xiii, 6, 19, 36, 37

[3] N. D. Academy, "nrf connect sdk fundamentals," 2024. [Online]. Available: https://academy.nordicsemi.com/courses/nrf-connect-sdk-fundamentals/ xvi, 34, 42, 89, 90, 95, 96

[4] M. M. Adame and N. Ameha, "Review on egg handling and management of incubation and hatchery environment," *Asian Journal of Biological Sciences*, 2023. [Online]. Available: https://doi.org/10.17311/ajbs.2023.474.484 4, 10, 11, 12

[5] OpenAI, "Chatgpt," 2024. [Online]. Available: https://chatgpt.com/ 4

[6] Grammarly, "Grammarly," 2024. [Online]. Available: https://www.grammarly.com/ 4

[7] Microsoft, "Visual studio code," 2024. [Online]. Available: https://code.visualstudio.com/ 6, 36, 66

[8] nRF Connect SDK, "Get started with nrf connect sdk - nordicsemi.com," 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-software/nRF-Connect-SDK/GetStarted?lang=en#infotabs 6, 16

[9] Zephyr, "Zephyr project – a proven rtos ecosystem, by developers, for developers," 2024. [Online]. Available: https://zephyrproject.org/ 6, 36

[10] MongoDB, "Mongodb: The developer data platform," 2024. [Online]. Available: https://www.mongodb.com/ 6, 20, 37

[11] I. Guevara, S. Ryan, A. Singh, C. Brandon, and T. Margaria, "Edge iot prototyping using model-driven representations: A use case for smart agriculture," *Sensors 2024, Vol. 24, Page 495*, vol. 24, p. 495, 1 2024. [Online]. Available: https://www.mdpi.com/1424-8220/24/2/495/htmhttps://www.mdpi.com/1424-8220/24/2/495 7, 8, 20, 21, 37

[12] R. A. R. A. Mouha, "Internet of things (iot)," *Journal of Data Analysis and Information Processing*, vol. 09, pp. 77–101, 4 2021. [Online]. Available: //file.scirp.org/html/3-2870386_108574.htmhttp://www.scirp.org/journal/Paperabs.aspx?PaperID=108574 7, 8

[13] M. Javaid, A. Haleem, I. H. Khan, and R. Suman, "Understanding the potential applications of artificial intelligence in agriculture sector," *Advanced Agrochem*, vol. 2, pp. 15–30, 3 2023. 9

[14] S. El-Gendy, "Iot based ai and its implementations in industries," *2020 15th International Conference on Computer Engineering and Systems (ICCES)*, 2020. 9

[15] S. G. Tzafestas, "Synergy of iot and ai in modern society: The robotics and automation case," *Robot Autom Eng J*, vol. 3, 2018. [Online]. Available: https://www.semanticscholar.org/paper/Security-Issues- 9

[16] S. Boßelmann, M. Frohme, D. Kopetzki, M. Lybecait, S. Naujokat, J. Neubauer, D. Wirkner, P. Zweihoff, and B. Steffen, "Dime: A programming-less modeling environment for web applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9953 LNCS, pp. 809–832, 2016. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-47169-3_60 18

[17] N. Semiconductor, "nrf command line tools," 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nRF-Command-Line-Tools/Download 35

[18] N. Semiconductor, "nrf connect for desktop," 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop/Download?lang=en#infotabs 35, 64

[19] embeddedguy, "Roadtest review: Nordic semiconductor thingy:53 development platform - element14 community," 2023. [Online]. Available: https://community.element14.com/products/roadtest/rv/roadtest_reviews/1672/nordic_thingy53_with_edge-impulse_review_document 39

[20] B. Sensortec, "Gas sensor bme688," 2024. [Online]. Available: https://www.bosch-sensortec.com/products/environmental-sensors/gas-sensors/bme688/ 41

[21] M. ul Hassan, "How to read nordic thingy:53 onboard bme688 sensor," 2022. [Online]. Available: https://www.hackster.io/mahmood-ul-hassan/how-to-read-nordic-thingy-53-onboard-bme688-sensor-003560 41

[22] W. Wassim, "Indoor air quality with thingy53 and bme688," 2023. [Online]. Available: https://www.hackster.io/wassfila/indoor-air-quality-with-thingy53-and-bme688-5c763b 42

[23] W. Wassim, "Home smart mesh - sdk-hsm-thingy53: Zephyr samples for nordic thingy53 dev kit openthread mesh with json data, bme688 bsec2 for air quality, light color, rgb les, battery," 2024. [Online]. Available: https://github.com/HomeSmartMesh/sdk-hsm-thingy53 42

[24] R. Semiconductor, "Bh1749nuc digital 16bit serial output type color sensor ic," 2024. [Online]. Available: https://www.rohm.com/products/sensors-mems/color-sensor-ics/bh1749nuc-product#productDetail 42

[25] M. ul Hassan, "How to read nordic thingy:53 onboard bh1749 light sensor," 2022. [Online]. Available: https://www.hackster.io/mahmood-ul-hassan/how-to-read-nordic-thingy-53-onboard-bh1749-light-sensor-bc591f 42

Appendixes

Exemplary Code

Read BME688 Sensor: Temperature, Humidity, Pressure, and Gas

Source Code

prj.conf

The `prj.conf` file that is given is a configuration file utilized in Zephyr OS projects to specify different settings and options for the project. The following is an explanation of each line of the configuration.

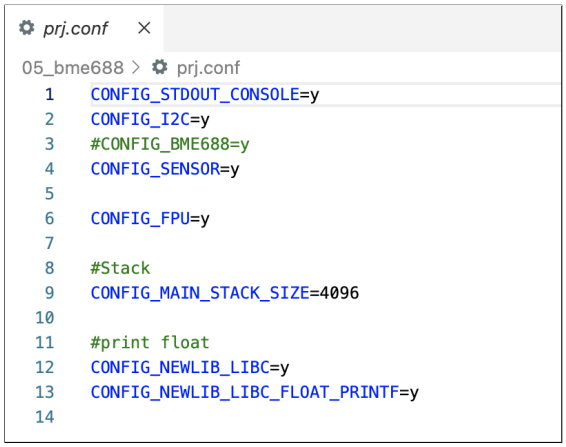


Figure 7.1: prj.conf - Read BME688 Sensor

- `CONFIG_STDOUT_CONSOLE=y`

This option redirects the standard output (`stdout`) to the console, displaying `printf` or similar output commands on the console.

- `CONFIG_I2C=y`

  This option enables the project to include support for the I2C (Inter-Integrated Circuit) protocol. I2C is frequently utilized to establish communication between the microcontroller and peripheral sensors such as the BME688.

- `CONFIG_SENSOR=y`

  This setting enables support for generic sensors in the project, allowing the utilization of different sensors integrated with the Zephyr sensor API.

- `CONFIG_FPU=y`

  Enable the microcontroller's Floating-Point Unit (FPU) to handle floating-point calculations efficiently. This is especially beneficial for tasks that require complex mathematical operations, such as processing sensor data.

- `CONFIG_MAIN_STACK_SIZE=4096`

  This command establishes the main thread's stack size at 4096 bytes. Defining the stack size is crucial for allocating the necessary memory for the main thread's execution, particularly when dealing with intricate tasks or extensive data structures, to prevent stack overflow errors.

- `CONFIG_NEWLIB_LIBC=y`

  Enabling the Newlib C library provides essential functionalities such as input/output processing, string manipulation, and mathematical operations. This library is a lightweight implementation of the C standard library.

- `CONFIG_NEWLIB_LIBC_FLOAT_PRINTF=y`

  This setting enables the capability to print floating-point numbers using the `printf` functions in the Newlib C library. Printing floating-point numbers (such as sensor temperatures) may not produce correct results if this setting is not enabled.

app.overlay

The file `app.overlay` is utilized in Zephyr OS to specify the particular application's hardware configurations and device settings. Normally, this file supplements or modifies certain sections of the device tree, which provides information about the hardware and peripherals accessible on a system. Here is an explanation of the given `app.overlay`.

Figure 7.2: app.overlay - Read BME688 Sensor

- `&i2c1 { ... }`

  This section defines the configuration for the I2C bus `i2c1`.

- `compatible = "nordic,nrf-twim";`

  This indicates that the `i2c1` interface is compliant with Nordic Semiconductor's TWIM (Two-Wire Interface Master) driver, which is the I2C master mode interface for nRF microcontrollers.

- `status = "okay";`

  This line defines that the I2C1 interface is enabled and needs to be initialized by the system.

- `clock-frequency = <I2C_BITRATE_FAST>;`

  This statement establishes the I2C bus speed at a "fast" bitrate. The `I2C_BITRATE_FAST` usually represents 400 kHz, which exceeds the standard 100 kHz.

- `pinctrl-0 = <&i2c1_default>;`

  This line defines the pin control configurations for the I2C1 interface's standard operating mode. The reference `&i2c1_default` probably indicates preconfigured pins in the hardware arrangement.

- `pinctrl-1 = <&i2c1_sleep>;`

  This line defines the pin control configurations for the I2C1 interface's sleep mode. The reference to `&i2c1_sleep` probably indicates the settings for when the device is in a low-power state.

- `pinctrl-names = "default", "sleep";`

  This assigns names to the pin control configurations defined above, linking them to `"default"` and `"sleep" modes`.

- `bmm150: bmm150@10 { ... }`

  Describes the BMM150 sensor, which is a geomagnetic sensor created by Bosch. The line `compatible = "bosch,bmm150";` indicates that this device works with the BMM150 sensor driver. The line `reg = <0x10>;` specifies that the I2C address of the sensor is `0x10`.

- `bh1749@38 { ... }`

  Specifies the BH1749 sensor, a digital colour sensor from Rohm. The statement `compatible = "rohm,bh1749";` denotes that this device suits the BH1749 sensor driver. The indication `reg = <0x38>;` communicates that the sensor's I2C address is `0x38`. The definition
  `int-gpios = <&gpio1 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;` describes the GPIO interrupt pin utilized by the BH1749 sensor. `&gpio1` references GPIO port 1. `5` specifies the specific pin number on GPIO port 1. The configuration `GPIO_PULL_UP | GPIO_ACTIVE_LOW` sets the pin as pull-up and

active-low, indicating that it expects the signal to be low (`0V`) for an interrupt.

- `bme688@76 { ... }`

  This block of code defines the BME688 sensor, which is an environmental sensor made by Bosch and is capable of measuring temperature, humidity, pressure, and gas resistance. The line `compatible = "bosch,bme688";` indicates that this device is compatible with the BME688 sensor driver. Lastly, `reg = <0x76>;` specifies that the sensor's I2C address is `0x76`.

CMakeLists.txt

The `CMakeLists.txt` file is crucial in CMake-based projects as it outlines the build configuration, identifies source files, and establishes the required environment for compiling the project. This is how the given `CMakeLists.txt` can be understood.

CMakeLists.txt

05_bme688 > CMakeLists.txt

```
#
# Copyright (c) 2018 Bosch Sensortec GmbH
#
# SPDX-License-Identifier: Apache-2.0
#

cmake_minimum_required(VERSION 3.20.0)
#default board can be overriden by -b
set(BOARD thingy53_nrf5340_cpuapp)
set(DTC_OVERLAY_FILE app.overlay)

# add Zephyr package
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(test_bme680)

# Add application source files
target_sources(app PRIVATE src/main.c)

# Include the directory containing the header file
zephyr_include_directories(drivers/bme688)
zephyr_include_directories(drivers/BME68x-Sensor-API)

# add the c-file
target_sources(app PRIVATE drivers/bme688/bme688.c)
target_sources(app PRIVATE drivers/BME68x-Sensor-API/bme68x.c)
```

Figure 7.3: CMakeLists.txt - Read BME688 Sensor

- **Minimum CMake Version**

  `cmake_minimum_required(VERSION 3.20.0)` This line specifies the minimum version of CMake needed to build this project. For this project, a CMake version of 3.20.0 or higher is necessary.

- **Board and Overlay Settings**

  `set(BOARD thingy53_nrf5340_cpuapp)` Establishes the default target board for the project as the Thingy:53 with the nRF5340 CPU application core. You can override this setting by specifying a different board using the `-b` option in the build command. `set(DTC_OVERLAY_FILE app.overlay)` Indicates that the `app.overlay` file will be utilized as the Device Tree overlay. This file offers additional or overriding configurations for the hardware description.

- **Finding Zephyr Package**

  The following CMake command
  `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})` locates and includes the Zephyr OS package necessary for the application. It uses the `REQUIRED` keyword to ensure the build fails if Zephyr is not found. The `HINTS` clause specifies to CMake where to search for Zephyr, commonly utilizing the `ZEPHYR_BASE` environment variable that points to the Zephyr installation directory.

- **Project Definition**

  Define the project as `test_bme680` with the `project(test_bme680)` command. CMake utilizes the name `test_bme680` internally to manage the build procedure.

- **Adding Application Source Files**

  When you use `target_sources(app PRIVATE src/main.c)`, you indicate the specific source files that must be compiled for the application. This means that the `main.c` file located in the `src` directory is a private source file for the `app` target.

- **Including Header Directories**

  The commands `zephyr_include_directories(drivers/bme688)` and `zephyr_include_directories(drivers/BME68x-Sensor-API)` add the specified directories (`drivers/bme688` and `drivers/BME68x-Sensor-API`) to the

include path. This ensures the compiler can locate the header files in these directories during the build process.

- **Adding Additional C Source Files**

  The commands `target_sources(app PRIVATE drivers/bme688/bme688.c)` and `target_sources(app PRIVATE drivers/BME68x-Sensor-API/bme68x.c)` are used to incorporate additional C source files into the application target. The `bme688.c` file is in the `drivers/bme688` directory and the `bme68x.c` file contained in the `drivers/BME68x-Sensor-API` directory is added as a private source file for the `app` target. These files likely contain the coding implementation for the BME688 sensor driver and associated APIs.

main.c

The provided `main.c` file is designed for a Zephyr-based application that runs on an embedded system. Its purpose is to interact with the Bosch BME688 sensor on a Nordic Semiconductor Thingy:53. The code is organized to initiate the sensor, collect environmental data such as temperature, pressure, humidity, and gas resistance, and then display this data in a continuous loop. Here's an in-depth explanation of the code.

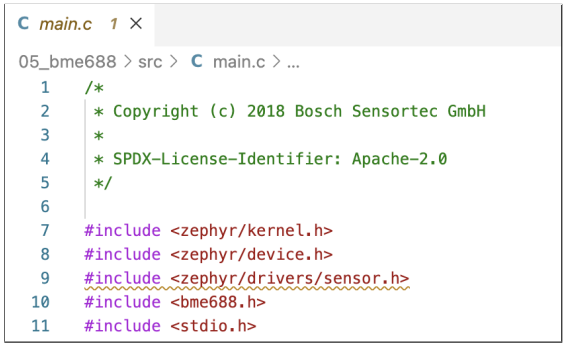


Figure 7.4: main.c part 1 - Read BME688 Sensor

- **Header Files**

  The `#include <zephyr/kernel.h>` statement brings in the kernel functions and data structures from the Zephyr RTOS, which are responsible for tasks

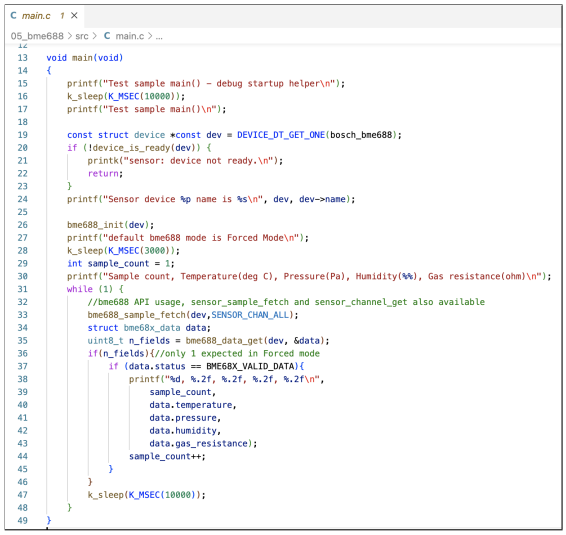


Figure 7.5: main.c part 2 - Read BME688 Sensor

like managing threads and implementing delays. To interact with devices in Zephyr, you can use `#include <zephyr/device.h>`. Additionally, the `#include <zephyr/drivers/sensor.h>` line provides access to the sensor driver API for interacting with various sensors. The `#include <bme688.h>` header is specific to the BME688 sensor and contains the necessary functions and structures to interface with it. Lastly, the `#include <stdio.h>` line brings in the standard C library for input/output operations, including functions like `printf`.

- **Main Function**

  `void main(void)` This serves as the application's entry point, and it is the standard starting function for a C program.
  `printf("Test sample main() - debug startup helper\n");`
  This line is responsible for printing a debug message to the console to indicate the start of the main function. `k_sleep(K_MSEC(10000));` This command pauses the execution for 10,000 milliseconds (10 seconds). This delay might be used for debugging purposes or to allow the system to stabilize before proceeding. `printf("Test sample main()\n");` This line prints another debug message to indicate that the main execution has resumed after the initial delay.

- **Device Initialization**

  `const struct device *const dev = DEVICE_DT_GET_ONE(bosch_bme688);` Retrieves a reference to the BME688 sensor device using the `DEVICE_DT_GET_ONE` macro, which is linked to the device tree entry for the BME688.
  `if (!device_is_ready(dev))` Checks if the device is ready for use. If not, it prints an error message and exits the function.
  `printf("Sensor device %p name is %s\n", dev, dev->name);` Prints the memory address and name of the sensor device to the console.

- **BME688 Sensor Initialization**

  `bme688_init(dev);` Initializes the BME688 sensor using default settings. Typically, this will set up the sensor to function in a specific mode (for example, Forced Mode). `printf("default bme688 mode is Forced Mode\n");` Outputs a message indicating that the sensor is in Forced Mode by default. `k_sleep(K_MSEC(3000));` Introduces a 3-second pause, perhaps to ensure the sensor is fully ready before data gathering starts.

- **Data Sampling and Output Loop**

  `int count = 1;` Initializes a counter to keep track of the number of samples taken. `printf("Sample count, Temperature(deg C), Pressure(Pa), Humidity(%), Gas resistance(ohm)\n");` Prints a header for the data that will be displayed, indicating the type of measurements. `while (1) { ... }` Initiates an endless loop to collect and display sensor data continuously. `bme688_sample_fetch(dev,SENSOR_CHAN_ALL);` Activates the sensor to take a measurement. `SENSOR_CHAN_ALL` indicates that all relevant data channels (temperature, pressure, humidity, gas resistance) should be accessed. `struct bme68x_data data;` Declares a structure to hold the accessed sensor data.
  `uint8_t n_fields = bme688_data_get(dev, &data);` Retrieves the data from the sensor and stores it in the `data` structure. The function returns the number of valid data fields accessed. `if(n_fields)` Checks if valid data was accessed. In Forced Mode, only one set of data is expected.

`if (data.status == BME68X_VALID_DATA){ ... }` Confirms that the accessed data is valid. `printf("%d, %.2f, %.2f, %.2f, %.2f\n", ...);` Displays the sample number, temperature, pressure, humidity, and gas resistance values to the console. `count+;+` Increases the sample count. `k_sleep(K_MSEC(10000));` Waits for 10 seconds before taking the next measurement.

Output

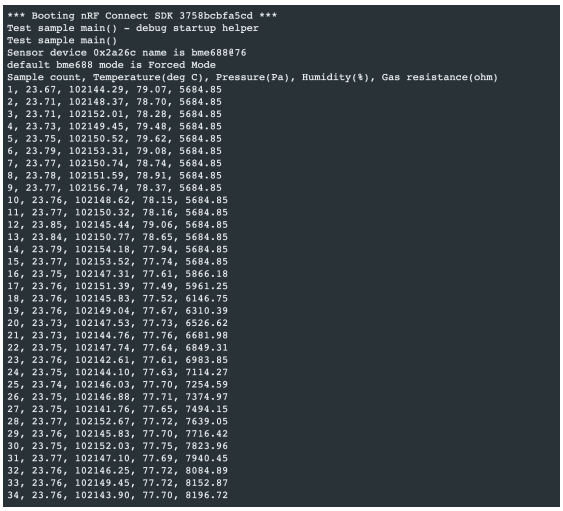


Figure 7.6: BME688 output with Bosch Sensortec's BME680 and BME688 sensor API

The figure 7.6 shows sensor readings from the Nordic Semiconductor Thingy:53 device, specifically from the BME688 sensor, which measures temperature, pressure, humidity, and gas resistance. This data was captured via a serial terminal during a test run. Here's an explanation of the output:

- **Booting nRF Connect SDK:**

  This indicates that the device has booted up using the nRF Connect SDK, the software development kit provided by Nordic Semiconductor for programming its IoT devices.

- **Test sample main() - debug startup helper:**

  This message is part of the test or debugging process, confirming that the primary function of the test sample has started. This is typical in embedded systems to ensure that the code execution is following the correct path.

- **Sensor device `0x2a26c` name is `bme688@76`:**

  This identifies the sensor in use, the BME688, which is connected to the system at I2C address `0x76`. The hexadecimal value `0x2a26c` is likely a memory address or device identifier used internally by the system.

- **default bme688 mode is Forced Mode:**

  The BME688 sensor operates in different modes, and "Forced Mode" is one in which the sensor takes a single measurement and then returns to sleep mode. This mode is energy efficient and suitable for periodic sampling.

- **Data Columns Explanation:** The data is presented in a structured format, with each row representing a single sample.

  - **Sample count:** The sequential number of each data sample (from 1 to 34 in provided data).
  - **Temperature (deg C):** The temperature reading in degrees Celsius. For example, in the first sample, the temperature is 23.67°C.
  - **Pressure (Pa):** The atmospheric pressure reading in Pascals. For instance, the first sample records 102144.29 Pa.
  - **Humidity (%)**: The relative humidity reading as a percentage. In the first sample, the humidity is 79.07%.
  - **Gas resistance (ohm):** This reading relates to air quality, with the BME688 measuring gas resistance in ohms. In the first sample, this is 5684.85 ohms. Higher values typically indicate cleaner air.

Data Interpretation

- **Consistency:** The temperature, pressure, and humidity readings are relatively stable across the 34 samples, suggesting a stable environment where the sensor operates.
- **Gas Resistance:** The gas resistance values remain nearly constant across the first 15 samples, with a slight increase in the final sample (from 5684.85 to 5866.18 ohms). This could indicate a slight change in the air quality, though the significance of this change would depend on the specific gases being detected.
- **Environmental Monitoring:** These sensor readings could be used in an egg-hatching environment to monitor and control critical conditions. Maintaining a stable temperature and humidity is crucial for optimal hatching rates, and monitoring gas levels could help ensure good air quality within the incubator.

When continuously monitored and analysed, this data can help make real-time adjustments to the hatching environment, optimising conditions for better hatch rates and reducing manual intervention.

Read BH1749: Digital Colour Sensor

Source Code

Kconfig

This `Kconfig` file configures the BH1749 sensor driver in a Zephyr project. It provides the capability to enable or disable the driver, configure triggers (potentially requiring GPIO), and set a wait duration for sensor initialization. Additionally, it offers the choice to activate an "Adaptive BH1749 Sensor," which may include specific or additional functionality compared to the standard BH1749 sensor. These choices enable developers to customize the sensor's behaviour to suit the specific requirements of their application. Here's an explanation of the content.

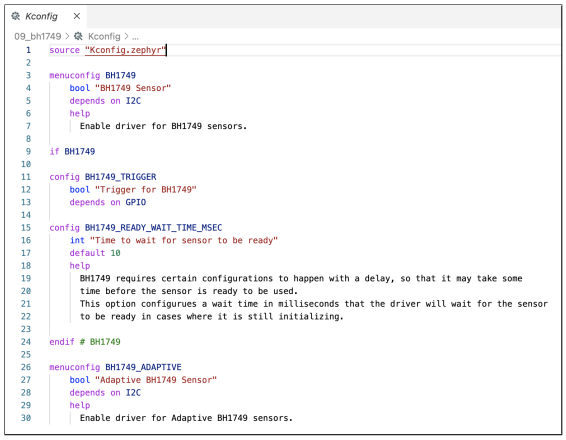


Figure 7.7: Kconfig - Read BH1749

- **Including Zephyr's Kconfig**

  Include `"Kconfig.zephyr"` from the source. This line incorporates the fundamental Zephyr Kconfig definitions. It imports the primary Zephyr configuration choices that this project will utilize.

- **Menu Configuration for BH1749 Sensor**

  `menuconfig BH1749` Generates a new menu configuration entry for the BH1749 sensor. Enabling this menu allows for configuring options related to the BH1749 sensor. `bool "BH1749 Sensor"` Defines a boolean option called "BH1749 Sensor." When chosen (`y`), it activates the BH1749 sensor driver. `depends on I2C` This option is accessible only if the I2C (Inter-Integrated Circuit) support is enabled, as the BH1749 sensor communicates via I2C. `help` Describes the option. In this instance, it states that enabling this option will activate the driver for BH1749 sensors.

- **Trigger Configuration for BH1749**

  `config BH1749_TRIGGER` Specifies a boolean configuration setting for activating or deactivating triggers for the BH1749 sensor.
  `bool "Trigger for BH1749"` If chosen (`y`), it activates trigger functionality for the BH1749 sensor. `depends on GPIO` This setting is accessible only if GPIO (General-Purpose Input/Output) support is enabled. The trigger functionality probably necessitates the use of GPIO pins.

- **Sensor Ready Wait Time Configuration**

  Define an integer configuration option (`config BH1749_READY_WAIT_TIME_MSEC`) to specify the duration the sensor needs to be ready. The option (`int "Time to wait for the sensor to be ready"`) enables the user to initialise a wait time (in milliseconds) for the BH1749 sensor. The default wait time is ten milliseconds. Additionally, the `help` function explains the necessity of this wait time. The user can configure the duration the driver should wait before the sensor is ready for use, as the BH1749 sensor requires a delay during initialization.

- **Menu Configuration for Adaptive BH1749 Sensor**

  `menuconfig BH1749_ADAPTIVE` Generates an additional menu configuration entry for an "Adaptive BH1749 Sensor". It seems to be a variation or specialized setup for the BH1749 sensor. `bool "Adaptive BH1749 Sensor"` Establishes a boolean option for activating the adaptive BH1749 sensor driver. `depends on I2C` This choice is accessible only if I2C support is enabled. `help`Offers an explanation that this choice activates the driver for an adaptive model of the BH1749 sensor.

app.overlay

The `app.overlay` file is responsible for setting up the I2C1 bus on the Nordic nRF53 microcontroller and specifying the three sensors that are connected to it: a magnetometer for plate number 1, a colour sensor for plate number 2, and an environmental sensor for plate number 3. Each sensor has its unique I2C address, and the colour sensor (plate number 2) also has a specific GPIO pin assigned for interrupt handling. These configurations ensure that the application software can access the sensors by loading the necessary drivers and properly initializing them to make them usable over the I2C bus. Here's a detailed breakdown.

- **I2C Bus Configuration** (`i2c1`)

app.overlay

09_bh1749 > app.overlay

```
1   &i2c1 {
2       compatible = "nordic,nrf-twim";
3       status = "okay";
4       clock-frequency = <I2C_BITRATE_FAST>;
5
6       pinctrl-0 = <&i2c1_default>;
7       pinctrl-1 = <&i2c1_sleep>;
8       pinctrl-names = "default", "sleep";
9       bmm150: bmm150@10 {
10          compatible = "bosch,bmm150";
11          reg = <0x10>;
12      };
13
14      bh1749@38 {
15          compatible = "rohm,bh1749";
16          reg = <0x38>;
17          int-gpios = <&gpio1 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
18      };
19
20      bme688@76 {
21          compatible = "bosch,bme688";
22          reg = <0x76>;
23      };
24  };
25
```

Figure 7.8: app.overlay - Read BH1749

The I2C1 peripheral on the microcontroller is denoted by `&i2c1`. The `&` symbol references an existing node in the base device tree. The node is compatible with the Nordic Semiconductor TWIM (Two-Wire Interface Master) driver, specified with `compatible = "nordic,nrf-twim"`. This driver controls the I2C1 peripheral. To enable the I2C1 peripheral, the status field is set to `"okay"`. The status field can have the value `"okay"` (to enable) or `"disabled"` (to disable). The I2C bus clock speed is set to `I2C_BITRATE_FAST`, typically corresponding to 400 kHz, with the `clock-frequency = <I2C_BITRATE_FAST>;` setting.

- **Pin Control Configuration**

  `pinctrl-0 = <&i2c1_default>;` Specifies the pin configuration for the I2C1 peripheral in its default (active) state. The reference `&i2c1_default` points to a predefined pin configuration setup. `pinctrl-1 = <&i2c1_sleep>;` Specifies the pin configuration for the I2C1 peripheral when it is in sleep mode. `pinctrl-names = "default", "sleep";` Associates names with the pin configurations, making it simpler to comprehend which configuration is applied in different power states.

- **BH1749 Color Sensor Configuration**

`bh1749@38` Creates a new node for a BH1749 colour sensor linked to the I2C1 bus and has the address `0x38`. The statement `compatible = "rohm,bh1749";` denotes the compatibility of this node with the ROHM BH1749 colour sensor driver. `reg = <0x38>;` specifies the I2C address for the BH1749 sensor, which is `0x38`. `int-gpios = <&gpio1 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;` indicates the GPIO pin used for receiving interrupts from the BH1749 sensor. This specifies that GPIO pin 5 on port 1 (`gpio1`) is utilized, with a pull-up resistor and active-low logic.

- **BME688 Environmental Sensor Configuration**

  The definition of a new node for an environmental sensor BME688 connected to the I2C1 bus at address `0x76` is given by `bme688@76`. The node is specified to be compatible with the Bosch BME688 sensor driver with the statement `compatible = "bosch,bme688";`. The I2C address for the BME688 sensor, which is `0x76`, is defined by `reg = <0x76>;`.

prj.conf

The `prj.conf` file is used to set up the Zephyr RTOS (Real-Time Operating System) project, configuring it to include necessary subsystems for I2C communication and sensor management, specifically for utilizing the digital colour sensor with the identifier BH1749. The logging system is also activated to allow the application to produce debug and information logs. The configuration is customized for an application that communicates with sensors via I2C, particularly the BH1749, and offers extra support for event-based triggers from the sensor. Here's a detailed explanation of each configuration option.

- `CONFIG_LOG=y`

  Enabling this feature activates the logging subsystem in Zephyr. When logging is enabled, the application can generate log messages, which are valuable for debugging and monitoring the application's performance. The logging system is a versatile framework that supports various logging levels (such as error, warning, and info) and can send logs to different destinations, such as consoles, files, or remote systems.

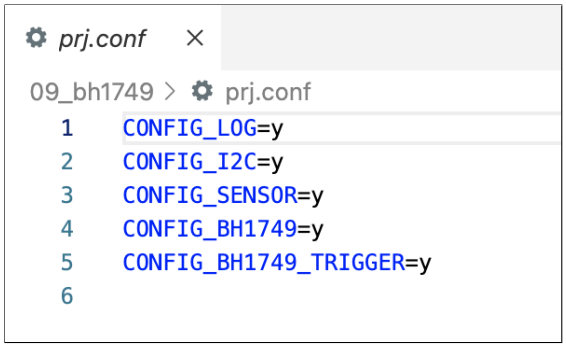


Figure 7.9: prj.conf - Read BH1749

- `CONFIG_I2C=y`

  Enabling this option in Zephyr supports the I2C (Inter-Integrated Circuit) subsystem, which is essential for any application communicating with devices over the I2C bus. The I2C bus connects sensors, memory chips, and peripherals to the microcontroller.

- `CONFIG_SENSOR=y`

  Enabling the generic sensor subsystem in Zephyr allows a unified interface for interacting with different sensor devices. This interface standardizes how applications interact with sensors, such as reading data, regardless of the specific sensor type.

- `CONFIG_BH1749=y`

  This option enables functionality for the driver of the digital colour sensor BH1749. It encompasses the essential driver required for interacting with the BH1749 colour sensor. It facilitates the application's access to information such as the intensity of red, green, blue, and infrared light that the sensor can measure.

- `CONFIG_BH1749_TRIGGER=y`

  This enables the activation of trigger support for the BH1749 sensor. When a specific event occurs (e.g., new data is ready to be read), triggers allow the sensor to notify the system. This process is usually managed through

interrupts, which enhances the system's efficiency by responding to events rather than continuously checking the sensor for data.

CMakeLists.txt

The `CMakeLists.txt` file is designed to compile a Zephyr project for the Thingy:53 development board and is specifically designed for the BH1749 sensor. The file specifies the board, incorporates a device tree overlay for hardware configuration, identifies the location of the Zephyr OS, defines the main source file, and includes a directory for additional header files. This setup enables the project to be compiled with the necessary configurations for interfacing with the BH1749 sensor on the designated hardware platform. Please find below a breakdown of the crucial components of this file.

CMakeLists.txt M

09_bh1749 > CMakeLists.txt

```
1
2  #
3  # Copyright (c) 2018 Bosch Sensortec GmbH
4  #
5  # SPDX-License-Identifier: Apache-2.0
6  #
7
8  cmake_minimum_required(VERSION 3.20.0)
9  #default board can be overriden by -b
10 set(BOARD thingy53_nrf5340_cpuapp)
11 set(DTC_OVERLAY_FILE app.overlay)
12
13 # add Zephyr package
14 find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
15 project(bh1749)
16
17 # Add application source files
18 target_sources(app PRIVATE src/main.c)
19
20 # Include the directory containing the header file
21 zephyr_include_directories(include)
22
```

Figure 7.10: CMakeLists.txt - Read BH1749

- `cmake_minimum_required(VERSION 3.20.0)`

  The project's build script requires at least CMake version 3.20.0 to ensure compatibility with the commands and features.

- `set(BOARD thingy53_nrf5340_cpuapp)`

The following line establishes the project's default board. It specifies the target board as `thingy53_nrf5340_cpuapp`, which represents the application core of the Nordic Semiconductor Thingy:53 development kit powered by the nRF5340 SoC. This board configuration will determine the hardware setup and settings utilized during the building process.

- `set(DTC_OVERLAY_FILE app.overlay)`

  An overlay file for the Device Tree is specified using the `app.overlay` file. This file includes extra or altered device tree details for the project, such as sensor connections, pin configurations, and other hardware-specific settings. This overlay file will be combined with the default device tree to tailor the hardware configuration during the build process.

- `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})`

  This command locates and adds the Zephyr OS package. It instructs CMake to find the Zephyr operating system necessary for project compilation. The `HINTS $ENV{ZEPHYR_BASE}` component ensures that the Zephyr installation path specified by the `ZEPHYR_BASE` environment variable is utilized.

- `project(bh1749)`

  The project is named "bh1749," which indicates that it likely revolves around the BH1749 digital colour sensor.

- `target_sources(app PRIVATE src/main.c)`

  The project specifies the source files to be compiled, including the `src/main.c` file as a source file for the application target is what this line does. This file contains the main C code where the application logic is implemented.

- `zephyr_include_directories(include)`

  Specifies extra directories with header files. This command instructs the compiler to add the `include` directory to the project's search path for header files. The `include` directory may hold specialized or sensor-specific headers essential for the project.

main.c

This `main.c` file is created to set up and communicate with the BH1749 colour sensor on a Zephyr-based system. It initializes the sensor to activate events according to data-ready signals or threshold breaches. After the configuration, it goes into an endless loop to retrieve and display the colour readings from the sensor. The program showcases the common usage of Zephyr's sensor API for real-time sensor data collection and event-based processing. Below is a comprehensive explanation of the code.

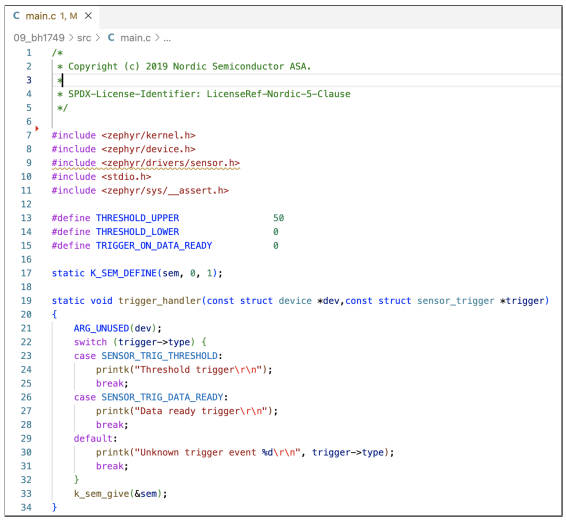


Figure 7.11: main.c part 1 - Read BH1749

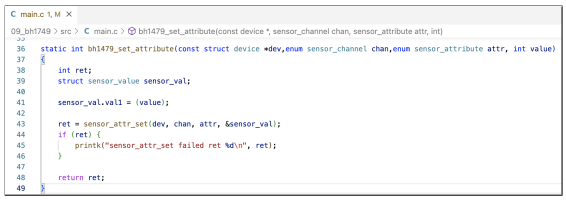


Figure 7.12: main.c part 2 - Read BH1749

- **Header Inclusions**

  `#include <zephyr/kernel.h>` This includes the kernel API, essential for thread management and synchronization tasks. `#include <zephyr/device.h>`

Figure 7.13: main.c part 3 - Read BH1749

This provides access to the device driver API.
`#include <zephyr/drivers/sensor.h>` This includes the sensor driver API, enabling interaction with sensor devices. `#include <stdio.h>` This is the standard C library for input/output functions. `#include <zephyr/sys/__assert.h>` This includes assert functions to aid in debugging.

- **Macros and Constants**

  The upper and lower thresholds for sensor readings are defined by `THRESHOLD_UPPER` and `THRESHOLD_LOWER`. These thresholds are used to set the sensor to activate an event when a reading exceeds these limits. The macro `TRIGGER_ON_DATA_READY` is currently set to `0`, indicating that the sensor will be activated based on threshold events rather than data-ready events.

- **Semaphore Initialization**

  `K_SEM_DEFINE(sem, 0, 1);` This line of code defines a semaphore named `sem`. The initial count of the semaphore is 0, and the maximum count is 1. The purpose of this semaphore is to synchronize the main thread with the sensor trigger events.

main.c 1, M

09_bh1749 > src > main.c > process(const device *)

```c
54  static void process(const struct device *dev)
82      while (1) {
83          if (IS_ENABLED(CONFIG_BH1749_TRIGGER)) {
84              /* Waiting for a trigger event */
85              k_sem_take(&sem, K_FOREVER);
86          } else {
87              /* Waiting some time for first fetch if trigger disabled*/
88              k_sleep(K_MSEC(1000));
89          }
90
91          ret = sensor_sample_fetch_chan(dev, SENSOR_CHAN_ALL);
92          /* The sensor does only support fetching SENSOR_CHAN_ALL */
93          if (ret) {
94              printk("sensor_sample_fetch failed ret %d\n", ret);
95              return;
96          }
97
98          ret = sensor_channel_get(dev, SENSOR_CHAN_RED, &temp_val);
99          if (ret) {
100             printk("sensor_channel_get failed ret %d\n", ret);
101             return;
102         }
103         printk("BH1749 RED: %d\n", temp_val.val1);
104
105         ret = sensor_channel_get(dev, SENSOR_CHAN_GREEN, &temp_val);
106         if (ret) {
107             printk("sensor_channel_get failed ret %d\n", ret);
108             return;
109         }
110         printk("BH1749 GREEN: %d\n", temp_val.val1);
111
112         ret = sensor_channel_get(dev, SENSOR_CHAN_BLUE, &temp_val);
113         if (ret) {
114             printk("sensor_channel_get failed ret %d\n", ret);
115             return;
116         }
117         printk("BH1749 BLUE: %d\n", temp_val.val1);
118
119         ret = sensor_channel_get(dev, SENSOR_CHAN_IR, &temp_val);
120         if (ret) {
121             printk("sensor_channel_get failed ret %d\n", ret);
122             return;
123         }
124         printk("BH1749 IR: %d\n", temp_val.val1);
125         k_sleep(K_MSEC(2000));
126     }
127 }
```

Figure 7.14: main.c part 4 - Read BH1749

- **Trigger Handler Function**

  The `trigger_handler` function is invoked when the sensor produces a trigger event. It displays the trigger type (either threshold or data-ready) and then frees the semaphore to indicate that an event has occurred.

- **Sensor Attribute Configuration Function**

  `bh1479_set_attribute` sets a sensor attribute, such as the upper or lower threshold. It uses the `sensor_attr_set` API to adjust the sensor's attributes.

- **Sensor Processing Function**

  The main function called `process` is responsible for configuring the sensor,

Figure 7.15: main.c part 5 - Read BH1749

setting up triggers, and continuously retrieving and printing sensor data. Depending on the `TRIGGER_ON_DATA_READY` macro, the sensor is configured to trigger either a threshold or a data-ready event. The `sensor_trigger_set` API establishes the trigger, with the `trigger_handler` function specified as the callback. Within an endless loop, the function retrieves the most recent sensor data using `sensor_sample_fetch_chan`, which gathers data from all sensor channels. It then utilizes `sensor_channel_get` to retrieve specific colour channel values (red, green, blue, and infrared) and print them.

- **Main Function**

  `main`: This serves as the program's starting point. Initializing the Device: The program tries to obtain the device structure for the sensor with the BH1749 using the `DEVICE_DT_GET_ONE` method. If the device is unprepared, it outputs an error message and exits. Executing the Process: If the sensor is prepared, it initiates the data acquisition and processing loop by calling the `process` function.

Output

The figure 7.16 shows data readings from the BH1749 light sensor on the Nordic Semiconductor Thingy:53 device. The sensor captures light intensity in different colour channels: Red, Green, Blue, and Infrared (IR). The data is triggered based on a threshold condition, meaning the sensor reports data when the light intensity crosses a pre-defined threshold. Below is an explanation of the output:

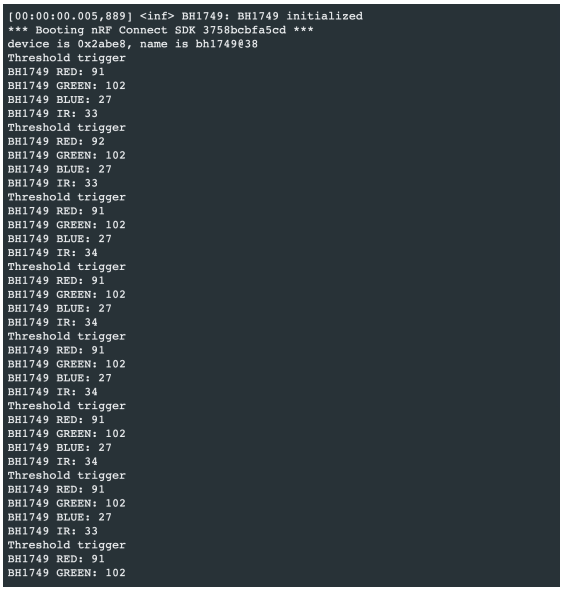


Figure 7.16: BH1749 Light Sensor output with Threshold trigger

- **Initialization:**

  The message `[00:00:00.005,889] <inf> BH1749: BH1749 initialized` indicates that the BH1749 sensor has been successfully initialized.

- **Device Identification:**

  `device is 0x2abe8, the name is bh1749@38`: The sensor is identified with a memory address `0x2abe8` and is connected to the system at I2C address `0x38`.

- **Threshold Trigger:**

  The repeated message `Threshold trigger` indicates that the light sensor has detected light levels that have crossed a predefined threshold, which triggers the device to capture and report the light intensity values.

- **Light Sensor Readings:**

  The output shows the intensity of light in different wavelengths or channels. The "RED" parameter represents the intensity of red light detected by the sensor, while the intensity of green light detected is represented by

the "GREEN" parameter. Similarly, the "BLUE" parameter indicates the intensity of blue light detected, and the "IR" parameter represents the intensity of infrared light detected. The specific readings for each colour channel in each trigger event are shown. For example, in the First Trigger, the raw sensor readings of light intensity for each respective channel are as follows: RED: 91, GREEN: 102, BLUE: 27, IR: 33. These values provide insight into the raw sensor readings of light intensity for each respective channel.

Data Interpretation

- **Consistency:** The Red, Green, Blue, and IR channel readings are fairly consistent across multiple threshold triggers. This suggests a stable light environment where light intensities don't vary significantly.

- **Triggering Mechanism:** The output indicates that the sensor is configured to report data when the light intensities cross a certain threshold. However, since the values are consistent, this threshold might be set very close to the ambient light levels, causing frequent triggering.

- **Environmental Monitoring:** In a practical application like an egg-hatching environment, these light sensor readings could be used to monitor and control lighting conditions, which is essential for developing embryos in eggs. Consistent lighting conditions can be critical, and any significant changes (e.g., an unexpected drop in light levels) could trigger alerts or adjustments.

- **Data Utilization:** The sensor data could be further analysed to detect trends or patterns in light conditions over time. This could help optimise the lighting environment to improve hatch rates or the overall health of the eggs.

The data shows that the BH1749 sensor effectively monitors light levels and triggers data capture when certain thresholds are met. The consistent readings indicate stable conditions crucial for maintaining optimal environmental parameters in controlled settings like egg hatcheries.

Read BH1749 and BME688 Sensors

Source Code

Kconfig

This `Kconfig` file contains configuration settings that allow for the activation and configuration of drivers for the BME688 and BH1749 sensors within a Zephyr project. The file provides options to activate the sensors, configure trigger mechanisms, and establish specific initialization parameters. These selections are fully integrated into Zephyr's comprehensive configuration system and can be customized by the user throughout the build process. Below is a breakdown of each component.

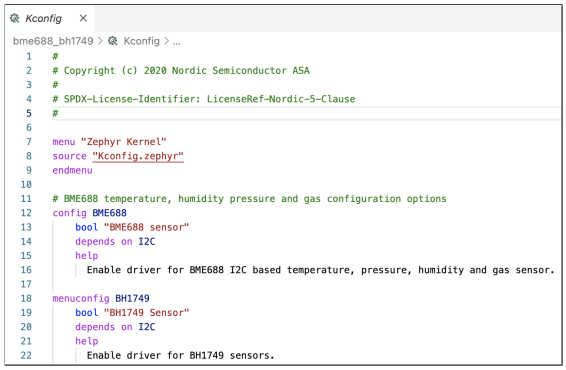


Figure 7.17: Kconfig part 1 - Read BH1749 and BME688

Figure 7.18: Kconfig part 2 - Read BH1749 and BME688

- **Zephyr Kernel Configuration Menu**

  The lines `menu "Zephyr Kernel"` and `endmenu` define a menu in the configuration system that groups various kernel-related options. The content within this menu is taken from another file. The line `source "Kconfig.zephyr"` imports the Zephyr kernel configuration options from the main `Kconfig.zephyr` file, allowing these options to be included in the current configuration menu.

- **BME688 Sensor Configuration**

  `config BME688` Establishes a boolean configuration setting for the BME688 sensor, encompassing temperature, humidity, pressure, and gas measurements. `bool "BME688 sensor"` This setting permits the user to activate or deactivate the BME688 sensor driver. It is presented as a checkbox in the configuration menu. `depends on I2C` This setting is only accessible if I2C support is enabled, as the BME688 communicates via the I2C bus. `help` Offers a brief description of the impact of enabling this setting- specifically, activating the driver for the BME688 sensor.

- **BH1749 Sensor Configuration**

  `menuconfig BH1749` sets up a menu configuration option for the BH1749 digital colour sensor. `bool "BH1749 Sensor"` allows the user to activate or deactivate the BH1749 sensor driver, similar to the BME688 option. `depends on I2C` This option is reliant on the activation of I2C support because the BH1749 sensor is also I2C-based. `help` gives a concise explanation that enabling this option activates the driver for the BH1749 sensor.

- **Conditional Configuration for BH1749**

  `if BH1749` This line initiates a conditional block containing options only if the BH1749 sensor option is activated. `config BH1749_TRIGGER` This setting activates the trigger functionality for the BH1749 sensor.
  `bool "Enable trigger for BH1749"` This boolean option allows the user to activate or deactivate the trigger functionality. `depends on GPIO` This setting relies on GPIO support, as triggers typically utilize GPIO lines for

signalling. `config BH1749_READY_WAIT_TIME_MSEC` This configuration establishes the wait time for the sensor to be ready after specific operations. `int "Sensor readiness wait time"` This integer option enables the user to define the wait time in milliseconds. `default 10` Establishes the default wait time as 10 milliseconds. `help` Clarifies that the BH1749 sensor may require initialization time after specific configurations, and this option determines how long the driver should wait. `endif # BH1749` Concludes the conditional block that relies on the BH1749 sensor being activated.

- **Adaptive BH1749 Sensor Configuration**

  `menuconfig BH1749_ADAPTIVE` Introduces a separate menu configuration for an "Adaptive" iteration of the BH1749 sensor. `bool "Adaptive BH1749 Sensor"` This setting activates or deactivates the adaptive mode for the BH1749 sensor. `depends on I2C` This setting is also reliant on I2C support. `help` Offers an explanation that this setting enables the driver for an adaptive version of the BH1749 sensor.

app.overlay

The file `app.overlay` sets up sensors BH1749 and BME688 to connect to the I2C1 bus on the nRF5340 SoC. Each sensor is assigned a specific I2C address, and the BH1749 sensor is also set up with an interrupt line. This overlay adjusts the default hardware configuration to ensure the device's software correctly recognizes and interfaces with these sensors.

- `&i2c1` **Node**

  Below is the modified text with reworded sentences:

  `&i2c1` denotes the I2C1 bus controller found on the nRF5340 SoC. The `&` symbol references an existing node in the base Device Tree. The line `compatible = "nordic,nrf-twim";` indicates that the I2C controller is compatible with Nordic Semiconductor's Two-Wire Interface Master (TWIM) driver. The line `status = "okay";` activates the I2C1 bus, which is necessary if the bus is disabled in the base Device Tree by default. The line

app.overlay 2 ×

bme688_bh1749 > app.overlay > ...

```
1   &i2c1 {
2       compatible = "nordic,nrf-twim";
3       status = "okay";
4       clock-frequency = <I2C_BITRATE_FAST>;
5
6       pinctrl-0 = <&i2c1_default>;
7       pinctrl-1 = <&i2c1_sleep>;
8       pinctrl-names = "default", "sleep";
9       bmm150: bmm150@10 {
10          compatible = "bosch,bmm150";
11          reg = <0x10>;
12      };
13
14      bh1749@38 {
15          compatible = "rohm,bh1749";
16          reg = <0x38>;
17          int-gpios = <&gpio1 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
18      };
19
20      bme688@76 {
21          compatible = "bosch,bme688";
22          reg = <0x76>;
23      };
24  };
```

Figure 7.19: app.overlay - Read BH1749 and BME688

`clock-frequency = <I2C_BITRATE_FAST>;` establishes the I2C bus clock speed to fast mode, generally at 400 kHz. The lines `pinctrl-0 = <&i2c1_default>;` and `pinctrl-1 = <&i2c1_sleep>;` define the pin control states for the I2C1 bus. `pinctrl-0` specifies the default configuration when the I2C bus is active, while `pinctrl-1` describes the configuration when the bus is in sleep mode. The line `pinctrl-names = "default", "sleep";` gives names to the pin control states corresponding to the configurations defined above.

- **bh1749 Sensor Configuration**

  `bh1749@38` Describes a node for the digital colour sensor named BH1749. The `@38` denotes that the sensor's I2C address is `0x38`. The statement `compatible = "rohm,bh1749";` indicates that the sensor is in line with the BH1749 driver from Rohm. The line `reg = <0x38>;` communicates that the sensor's I2C address is `0x38`. The `int-gpios = <&gpio1 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;` sets up an interrupt line for the BH1749 sensor. This line is linked to GPIO pin 5 on GPIO port 1. The `GPIO_PULL_UP` flag activates a pull-up resistor on this line, and `GPIO_ACTIVE_LOW` signifies that a low signal triggers the interrupt.

- **bme688 Sensor Configuration**

The node `bme688@76` defines the BME688 sensor, with `@76` indicating the sensor's I2C address as `0x76`. The statement
`compatible = "bosch,bme688";` denotes that the sensor is compatible with the BME688 driver from Bosch. Additionally, `reg = <0x76>;` specifies the sensor's I2C address as `0x76`.

prj.conf

The `prj.conf` file within a Zephyr project contains various configuration options that manage the behaviour of the build system, the operating system, and the application. This configuration file establishes a Zephyr project that utilizes C++17, integrates support for particular sensors (BME688 and BH1749), and implements a reliable logging system for debugging purposes. Additionally, it sets up floating-point printing and ensures that the main thread has a sufficient stack size for execution, which is particularly crucial when using C++ and handling complex tasks. The compiler is also set to suppress specific warnings associated with ABI changes. Now, let's go through each configuration option in the `prj.conf` file.

prj.conf

bme688_bh1749 > prj.conf > ...

```
1  #Sensor
2  CONFIG_STDOUT_CONSOLE=y
3  CONFIG_I2C=y
4  CONFIG_BME688=y
5  CONFIG_BH1749=y
6  CONFIG_BH1749_TRIGGER=y
7  CONFIG_SENSOR=y
8  #CONFIG_FPU=y
9
10 #C++
11 CONFIG_CPLUSPLUS=y
12 CONFIG_LIB_CPLUSPLUS=y
13 CONFIG_EXCEPTIONS=y
14 CONFIG_NEWLIB_LIBC=y
15 CONFIG_NEWLIB_LIBC_NANO=n
16
17 CONFIG_STD_CPP11=n
18 CONFIG_STD_CPP17=y
19
20 #Stack
21 CONFIG_MAIN_STACK_SIZE=4096
22
23 #safe silence of ABI change warning as using gcc > 7 https://github.com/nlohmann/json/issues/1861
24 CONFIG_COMPILER_OPT="-Wno-psabi"
25
26 #print float
27 CONFIG_NEWLIB_LIBC_FLOAT_PRINTF=y
28
29 # Logging configuration
30 CONFIG_LOG=y
31 #CONFIG_LOG_DEFAULT_LEVEL=4
```

Figure 7.20: prj.conf - Read BH1749 and BME688

- **Sensor Configuration**

  `CONFIG_STDOUT_CONSOLE=y` This setting enables directing standard output (e.g., `printf`) to the console, which is crucial for debugging and logging.

`CONFIG_I2C=y` This setting supports the I2C (Inter-Integrated Circuit) protocol, allowing the microcontroller to communicate with I2C devices such as sensors. `CONFIG_BME688=y` This setting enables the driver for the Bosch BME688 sensor, which is capable of measuring temperature, humidity, pressure, and gas. `CONFIG_BH1749=y` This setting enables the driver for the Rohm BH1749 colour sensor. `CONFIG_BH1749_TRIGGER=y` This setting enables trigger support for the BH1749 sensor, allowing it to generate interrupts based on specific conditions such as data readiness or threshold events. `CONFIG_SENSOR=y` This setting enables the Zephyr Sensor API, which provides a common interface for interacting with different sensors.

- **C++ Configuration**

  C++ support is enabled in the project by `CONFIG_CPLUSPLUS=y`, allowing the use of C++ code alongside C. The C++ standard library is enabled by `CONFIG_LIB_CPLUSPLUS=y`, providing access to standard C++ features. Exception handling in C++ (`try`, `catch`, `throw`) is enabled by `CONFIG_EXCEPTIONS=y`. The Newlib C library, a lightweight C standard library for embedded systems, is enabled by `CONFIG_NEWLIB_LIBC=y`. The "nano" version of Newlib, which is even more lightweight but lacks some features, is disabled by `CONFIG_NEWLIB_LIBC_NANO=n`. C++11 standard support is disabled by `CONFIG_STD_CPP11=n`. C++17 standard support, which includes more modern features like improved performance, language improvements, and new libraries, is enabled by `CONFIG_STD_CPP17=y`.

- **Stack Configuration**

  `CONFIG_MAIN_STACK_SIZE=4096` Establishes the main thread's stack size as 4096 bytes. This memory allocation is vital when utilizing C++ capabilities and managing intricate tasks.

- **Compiler and Build Configuration**

  `CONFIG_COMPILER_OPT="-Wno-psabi"` This adds a compiler option to ignore warnings related to changes in the Application Binary Interface (ABI). It's especially important when using GCC versions higher than 7, as there could be ABI compatibility issues.

- **Floating Point Support**

  Enabling support for printing floating-point numbers (e.g., `printf("%f")`) in Newlib is achieved by setting `CONFIG_NEWLIB_LIBC_FLOAT_PRINTF` to `y`. This feature is commonly deactivated in embedded systems by default to conserve space.

- **Logging Configuration**

  Enabling `CONFIG_LOG=y` provides logging support in Zephyr. This enables the application to produce valuable log messages for debugging and monitoring purposes. Although currently commented out, `#CONFIG_LOG_DEFAULT_LEVEL=4` would establish the default log level as 4 (Debug level) if enabled, offering detailed information about the program's execution.

CMakeLists.txt

The `CMakeLists.txt` file is created to set up and compile a Zephyr-based project for the Thingy:53 development board. It contains configurations for compiling C and C++ sources, includes necessary directories, and adds or removes source files and directories based on the configuration options set in `prj.conf`. The project is designed to be compatible with the BME688 and BH1749 sensors, incorporating their drivers and API into the building process. Here is an explanation of each section of the script.

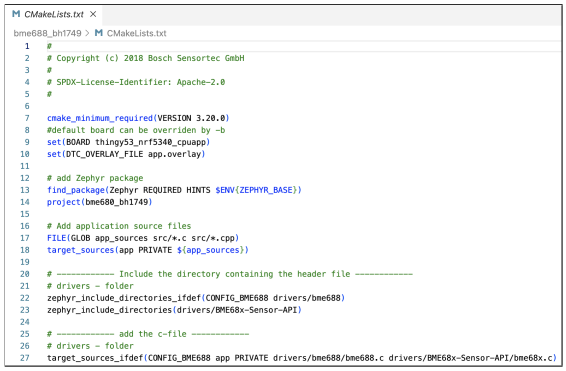


Figure 7.21: CMakeLists.txt - Read BH1749 and BME688

- **Basic Configuration**

  The `cmake_minimum_required(VERSION 3.20.0)` statement sets the minimum required CMake version for building the project to 3.20.0. This ensures that the project can utilize the features and fixes specific to this version of CMake. The `set(BOARD thingy53_nrf5340_cpuapp)` command designates the default target board for the project as `thingy53_nrf5340_cpuapp`, which corresponds to the Thingy:53 development board based on the Nordic nRF5340 chip. The `set(DTC_OVERLAY_FILE app.overlay)` command specifies the `app.overlay` Device Tree Overlay file to be utilized during the build process. Generally, this file includes customized hardware configuration details that override the default device tree settings.

- **Zephyr Package and Project Setup**

  `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})`. This command is used to search for the Zephyr OS package, which is necessary for this project. The `HINTS` keyword is utilized to assist in locating Zephyr's base directory by using the `ZEPHYR_BASE` environment variable. The command `project(bme680_bh1749)` defines the project name as `bme680_bh1749`. This name is utilized in various stages of the build process.

- **Source Files Configuration**

  In the `src` directory, the `FILE(GLOB app_sources src/*.c src/*.cpp)` command collects all the `.c` and `.cpp` source files and saves them in the `app_sources` variable. This is a convenient way to automatically include all relevant source files in the build without specifying them individually. Using `target_sources(app PRIVATE ${app_sources})`, the source files stored in the `app_sources` variable are added as private to the `app` target. This implies that these files will be compiled and linked as part of the `app` executable, but their definitions won't be accessible to other targets within the project.

- **Include Directories and Additional Source Files**

  `zephyr_include_directories_ifdef(CONFIG_BME688 drivers/bme688)` When the `CONFIG_BME688` option is enabled (from `prj.conf`), this command

includes the `drivers/bme688` directory in the list of included directories. This action ensures that the headers in this directory will be available during the project compilation.
`zephyr_include_directories(drivers/BME68x-Sensor-API)` This includes the `drivers/BME68x-Sensor-API` directory in the include path unconditionally, allowing the project to utilize headers from this directory.
`target_sources_ifdef(CONFIG_BME688 app PRIVATE drivers/bme688/bme688.c drivers/BME68x-Sensor-API/bme68x.c)` When `CONFIG_BME688` is enabled, this command appends the `bme688.c` and `bme68x.c` source files to the `app` target. These files are essential drivers required to support the BME688 sensor.

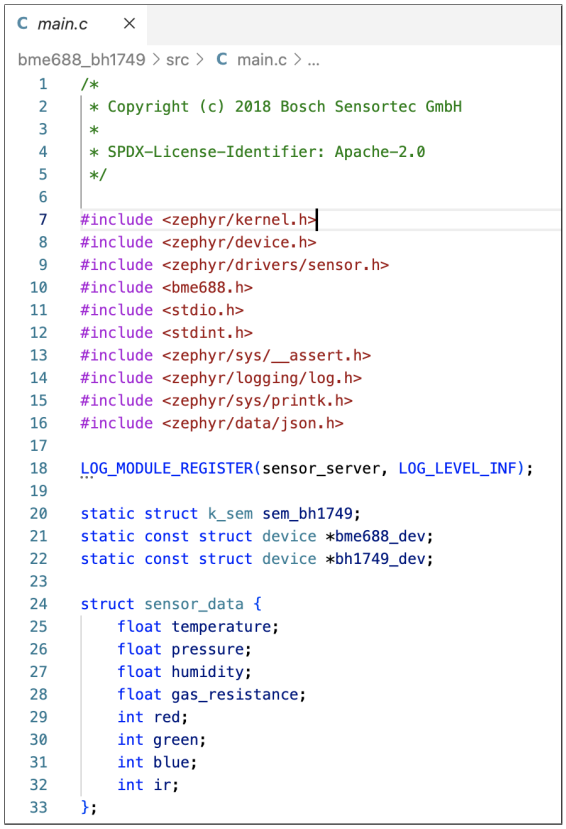
main.c

The `main.c` file retrieves environmental information (temperature, pressure, humidity, gas resistance) from a sensor labelled BME688 and collects colour and infrared data from another sensor labelled BH1749. It employs triggers to optimize the data retrieval process from the BH1749 sensor and periodically outputs the gathered data in JSON format to the console. This design enables efficient and structured sensor data handling in an embedded environment utilizing the Zephyr RTOS. Below is a detailed explanation of the functionality of the code.

- **Header Inclusions and Definitions**

  Zephyr Kernel and Device Libraries contain headers for kernel functions, device access, sensor drivers, and other utilities from the Zephyr OS. BME688 Sensor Library is designed for managing the Bosch BME688 sensor. The logging module is utilized for debugging messages, and JSON tools are included but not explicitly employed for JSON processing in this code snippet. The Log Module establishes a logging module called `sensor_server` with an information-level logging setting.

- **Data Structures and Global Variables**



main.c

bme688_bh1749 > src > main.c > ...

```c
/*
 * Copyright (c) 2018 Bosch Sensortec GmbH
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/drivers/sensor.h>
#include <bme688.h>
#include <stdio.h>
#include <stdint.h>
#include <zephyr/sys/__assert.h>
#include <zephyr/logging/log.h>
#include <zephyr/sys/printk.h>
#include <zephyr/data/json.h>

LOG_MODULE_REGISTER(sensor_server, LOG_LEVEL_INF);

static struct k_sem sem_bh1749;
static const struct device *bme688_dev;
static const struct device *bh1749_dev;

struct sensor_data {
    float temperature;
    float pressure;
    float humidity;
    float gas_resistance;
    int red;
    int green;
    int blue;
    int ir;
};
```

Figure 7.22: main.c part 1 - Read BH1749 and BME688

A custom data structure called `sensor_data` is used to store the readings from both sensors. The first sensor, BME688, measures temperature, pressure, humidity, and gas resistance, while the second sensor, BH1749, measures red, green, blue, and infrared (IR) light intensity.

- **Synchronization Mechanism**

  A synchronization mechanism called a semaphore (`sem_bh1749`) coordinates the retrieval of data from the BH1749 sensor when a triggering event occurs.

- **Trigger Handler and Sensor Initialization**

  When the BH1749 sensor detects that new data is available (using a trigger), the semaphore is released to indicate that the data is ready to be retrieved. This function initializes the BH1749 sensor to initiate data re-

Figure 7.23: main.c part 2 - Read BH1749 and BME688

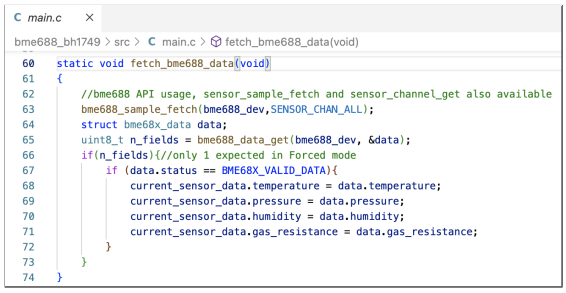


Figure 7.24: main.c part 3 - Read BH1749 and BME688

trieval when new data is available. It links the `trigger_handler` with the `SENSOR_TRIG_DATA_READY` event.

- **Data Fetching Functions**

  BME688 Retrieving Data This function obtains temperature, pressure, humidity, and gas resistance data from the BME688 sensor and saves it in the `current_sensor_data` struct. BH1749 Data Retrieval This function retrieves color data (red, green, blue) and infrared data from the BH1749 sensor. It waits for the semaphore (`sem_bh1749`) to indicate that the data is ready.

- **Main Function**

  During initialization, the main function starts by setting up the trigger for data readiness of the BH1749 sensor. The BME688 sensor is then initialized and set to Forced Mode, which is a mode where the sensor takes a single

main.c

bme688_bh1749 > src > main.c > fetch_bh1749_data(void)

```c
static void fetch_bh1749_data(void)
{
    struct sensor_value val;
    int ret;

    k_sem_take(&sem_bh1749, K_FOREVER);

    ret = sensor_sample_fetch(bh1749_dev);
    if (ret) {
        LOG_ERR("Failed to fetch BH1749 data: %d", ret);
        return;
    }

    sensor_channel_get(bh1749_dev, SENSOR_CHAN_RED, &val);
    current_sensor_data.red = val.val1;
    sensor_channel_get(bh1749_dev, SENSOR_CHAN_GREEN, &val);
    current_sensor_data.green = val.val1;
    sensor_channel_get(bh1749_dev, SENSOR_CHAN_BLUE, &val);
    current_sensor_data.blue = val.val1;
    sensor_channel_get(bh1749_dev, SENSOR_CHAN_IR, &val);
    current_sensor_data.ir = val.val1;
}
```

Figure 7.25: main.c part 4 - Read BH1749 and BME688

measurement and then returns to sleep. In the main loop, the application continuously retrieves data from both sensors. Following each data retrieval, the collected data is printed as a JSON-formatted string to the console. This loop repeats every 5 seconds.

Output

The data provided in figure 7.29 shows the output readings from two sensors on the Thingy:53 device, the BH1749 light sensor and the BME688 environmental sensor. The BME688 environmental sensor provides crucial environmental data. It measures Celsius (°C) temperature, providing insights into the ambient temperature. Additionally, it measures atmospheric pressure in Pascals (Pa), which is essential for weather monitoring. Humidity is measured in percentage (%), offering valuable information about the relative humidity in the air. Furthermore, the sensor provides gas resistance in Ohms, indicating the presence of volatile organic compounds (VOCs) or gases, thus giving insights into air quality.

On the other hand, the BH1749 light sensor offers a different set of valuable data. It measures the Red, Green, and Blue (RGB) values, providing information about light intensity in these three colour channels. Additionally, it measures the intensity of infrared light, providing further insights into the light environment.

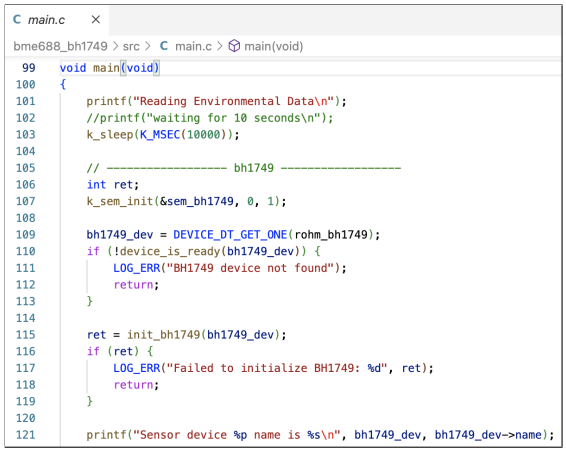


Figure 7.26: main.c part 5 - Read BH1749 and BME688

Figure 7.27: main.c part 6 - Read BH1749 and BME688

These sensors collectively offer a comprehensive understanding of the environmental conditions and light intensity, enabling analysis of trends over time.

Data Interpretation

Environmental Conditions (BME688):

- **Temperature:** The temperature readings are relatively stable, hovering around 24.0°C. This indicates a consistent ambient temperature in the monitored environment, which is crucial for egg-hatching.
- **Pressure:** The atmospheric pressure fluctuates slightly, ranging between 101413 Pa and 101438 Pa. Such minor variations are typical in a controlled indoor environment and might not significantly impact hatching conditions.
- **Humidity:** Humidity remains stable around 72-73%. This is an important parameter in egg hatching, as eggs require a specific humidity range for

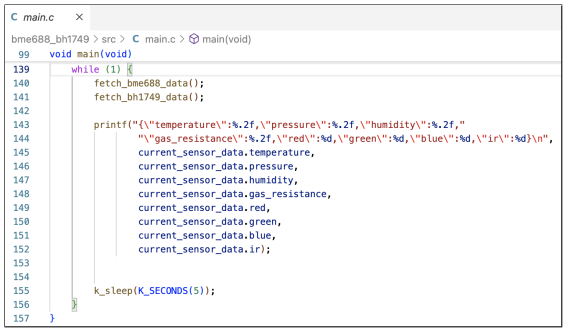


Figure 7.28: main.c part 7 - Read BH1749 and BME688

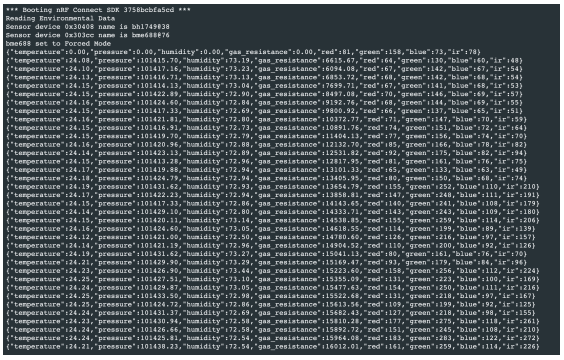


Figure 7.29: BH1749 and BME688 Sensors output

proper embryo development.

- **Gas Resistance:** The gas resistance value starts from around 6615 ohms and gradually increases to over 16500 ohms. An increasing gas resistance suggests a reduction in the presence of gases or VOCs in the environment, potentially indicating improving air quality. This is beneficial for maintaining a healthy hatching environment.

Light Conditions (BH1749):

- The light sensor readings show RGB and IR value variations. For example, the red value ranges from 64 to 183, green from 130 to 283, blue from 60 to 122, and IR from 48 to 272.
- These variations indicate changes in the lighting conditions within the environment, possibly due to changes in light source intensity or the positioning of the sensor relative to the light source.

- The significant jump in RGB and IR values towards the end of the readings suggests a notable change in light exposure, possibly due to an increase in light intensity or a change in the environment (e.g., the opening of a light source).

Implications for the Egg-Hatching Environment

- **Temperature and Humidity:** The consistent temperature and humidity levels are favourable for maintaining optimal conditions in an egg-hatching environment. These parameters are crucial for embryo development and successful hatching.

- **Air Quality (Gas Resistance):** The increasing gas resistance trend suggests improving air quality, which is beneficial as poor air quality can negatively impact hatching success rates.

- **Lighting Conditions:** The variability in light sensor readings indicates that the environment's lighting conditions are unstable. Consistent light exposure is important in a hatching environment, and fluctuations might need to be controlled or investigated further.

The data from the Thingy:53 device's sensors provides a comprehensive view of the environmental conditions crucial for egg hatching. The readings suggest that while stable temperature and humidity indicate a well-controlled environment, lighting fluctuations may require attention. The increase in gas resistance is a positive sign, indicating potential improvements in air quality over time. These insights could be used to optimise the egg-hatching process, ensuring better outcomes and operational efficiency.

Setting Up the USB Network Interface

Source Code

prj.conf

This `prj.conf` configuration is intended for a Zephyr-based firmware project that employs the USB Communications Device Class Abstract Control Model (CDC ACM) to establish a virtual serial port over USB. It also includes support for Ethernet-based networking. The setup allows IPv4 and IPv6 support and incorporates DHCP for dynamic IP address allocation. The logging settings are configured only to capture errors related to the USB components, and the UART communication is designed to be interrupt-driven for optimal performance. The USB device stack is not initialized during boot, so it must be manually initialized in the code. Below is an explanation of the specific settings provided. The following is an explanation of each line of the configuration.

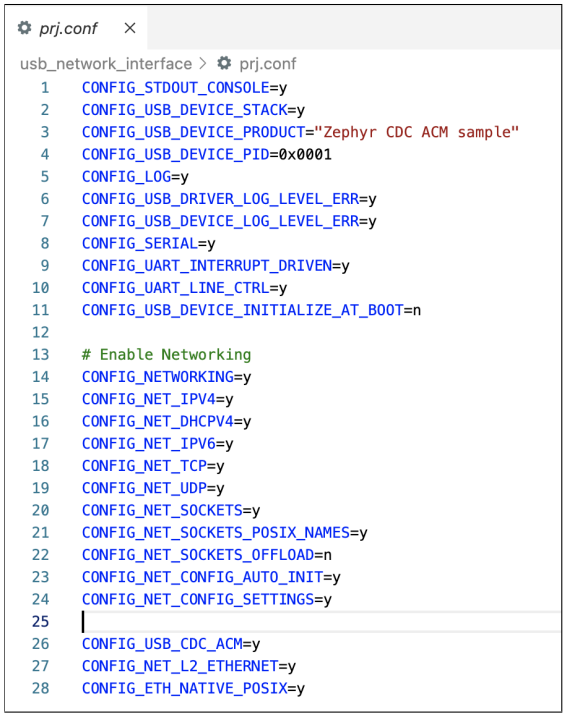


Figure 7.30: prj.conf Caption

- **Console and Logging**

  Enabling `CONFIG_STDOUT_CONSOLE=y` allows the system to utilize the console for standard output, enabling the printing of messages (e.g., logs) to the console. Enabling `CONFIG_LOG=y` activates Zephyr's logging subsystem,

which is used for debugging and informational output. Setting `CONFIG_USB_DRIVER_LOG_LEVEL_ERR=y` configures the logging level for the USB driver to "Error," ensuring that only error messages will be logged. Setting `CONFIG_USB_DEVICE_LOG_LEVEL_ERR=y` configures the logging level for the USB device stack to "Error."

- **USB Configuration**

  The USB device stack is enabled by setting `CONFIG_USB_DEVICE_STACK=y` in the firmware, allowing it to function as a USB device. The USB product name is set to "Zephyr CDC ACM sample" using `CONFIG_USB_DEVICE_PRODUCT="Zephyr CDC ACM sample"`. A unique identifier, the USB Product ID (PID), is assigned the value `0x0001` using `CONFIG_USB_DEVICE_PID=0x0001`. The automatic initialization of the USB device stack at boot is disabled by setting `CONFIG_USB_DEVICE_INITIALIZE_AT_BOOT=n`, requiring manual initialization in the application code. The USB CDC ACM (Communications Device Class Abstract Control Model), commonly employed for creating virtual serial ports over USB, is enabled through `CONFIG_USB_CDC_ACM=y`.

- **Serial and UART Configuration**

  Enabling `CONFIG_SERIAL=y` supports the serial driver, enabling communication over serial interfaces. Enabling `CONFIG_UART_INTERRUPT_DRIVEN=y` allows for more efficient and responsive UART operation through interrupt-driven communication instead of polling. Enabling `CONFIG_UART_LINE_CTRL=y` allows for the use of UART line control APIs, which are used to manage line-specific settings such as baud rate and parity.

- **Networking**

  `CONFIG_NETWORKING=y` Activates networking support, enabling the device to communicate over a network. `CONFIG_NET_IPV4=y` Activates IPv4 networking, allowing the device to utilize the IPv4 protocol for network communication. `CONFIG_NET_DHCPV4=y` Activates DHCPv4 support, enabling the device to obtain an IP address from a DHCP server automatically.

`CONFIG_NET_IPV6=y` Activates IPv6 networking, allowing the device to use the IPv6 protocol for network communication. `CONFIG_NET_TCP=y` Activates support for TCP (Transmission Control Protocol), which is crucial for reliable network communication. `CONFIG_NET_UDP=y` Activates support for UDP (User Datagram Protocol) for lightweight, connectionless network communication. `CONFIG_NET_SOCKETS=y` Activates the socket API, a common interface for network communication in C/C++.
`CONFIG_NET_SOCKETS_POSIX_NAMES=y` Activates POSIX-compliant socket names, making the socket API easier for those familiar with POSIX.
`CONFIG_NET_SOCKETS_OFFLOAD=n` Deactivates socket offloading, meaning that socket operations will be handled by the Zephyr networking stack rather than offloaded to specialized hardware. `CONFIG_NET_CONFIG_AUTO_INIT=y` Activates automatic initialization of network configuration, simplifying network setup. `CONFIG_NET_CONFIG_SETTINGS=y` Activates network configuration settings, likely including IP address, subnet mask, and gateway.

- **Ethernet and Networking Interface**

  Enabling Ethernet as the Link Layer (Layer 2) protocol with `CONFIG_NET_L2_ETHERNET=y` is essential for facilitating wired network communication. Enabling the native POSIX Ethernet driver with `CONFIG_ETH_NATIVE_POSIX=y` allows the Zephyr application to utilize Ethernet networking in a POSIX-compliant environment, frequently used for simulation or native builds on a desktop OS.

app.overlay

The `app.overlay` customizes the configuration of Thingy:53 to utilize the USB Device Controller (`zephyr_udc0`) for USB CDC ACM functionality. It essentially prepares the device to operate as a virtual serial port over USB, commonly employed for serial communication with a host computer. This setup is essential when configuring Thingy:53 to act as a USB network adapter, enabling it to communicate with the host system via USB while offering serial communication capabilities. Here's a breakdown of the `app.overlay` File.

app.overlay

usb_network_interface > app.overlay

```
/*
 * Copyright (c) 2021 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */

&zephyr_udc0 {
    cdc_acm_uart0 {
        compatible = "zephyr,cdc-acm-uart";
    };
};

```

Figure 7.31: app.overlay - Setting Up the USB Network Interface

- `&zephyr_udc0`

  This node in the device tree is associated with the USB Device Controller (`udc0`). The symbol `&` denotes a node previously defined in the device tree, probably in the SoC's or board's device tree file.

- `cdc_acm_uart0`

  Under the `zephyr_udc0` node, a new child node is being defined. The node name `cdc_acm_uart0` shows that this node will serve the purpose of the USB CDC ACM functionality, particularly for UART0, which is typically the first and primary UART interface.

- `compatible = "zephyr,cdc-acm-uart";`

  This line specifies the compatibility of the "zephyr,cdc-acm-uart" driver with the `cdc_acm_uart0` node. Zephyr is instructed to utilize its CDC ACM UART driver for this particular node through this compatibility string. As a result, the device can appear as a USB CDC ACM device.

CMakeLists.txt

The `CMakeLists.txt` file is the initial setup for building a Zephyr project with USB CDC ACM capabilities. It commences by locating and configuring Zephyr, integrating standard USB settings from the Zephyr SDK, and then including all necessary source files for the application. The project's primary goal is to create

an application named `cdc_acm`, which will likely require configuring the device to function as a USB CDC ACM device, enabling it to function as a virtual serial port over USB. Here's a breakdown of the code.

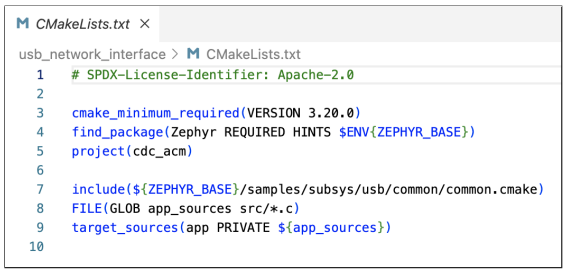


Figure 7.32: CMakeLists.txt - Setting Up the USB Network Interface

- `cmake_minimum_required(VERSION 3.20.0)`

  The minimum version of CMake required is 3.20.0 to ensure that the project will only be built using a CMake version that supports all the features utilized in this configuration.

- `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})`

  This instruction locates the Zephyr SDK package, essential for building the application. By using the `HINTS` option, CMake is directed to the `ZEPHYR_BASE` environment variable, which is expected to hold the location of the Zephyr base directory. This is essential for accessing Zephyr's build system and libraries.

- `project(cdc_acm)`

  This defines the project name as `cdc_acm`. CMake uses the project name internally to manage the build process and generate the necessary build files.

- `include(${ZEPHYR_BASE}/samples/subsys/usb/common/common.cmake)`

  The Zephyr SDK includes a CMake file (`common.cmake`) used in the USB subsystem samples. This `common.cmake` file probably has extra configuration settings, functions, or macros to configure and correctly build a USB CDC ACM application.

- `FILE(GLOB app_sources src/*.c)`

  Using `GLOB`, this instruction gathers all the C source files (`*.c`) from the `src` directory and saves them in the `app_sources` variable. This enables you to conveniently include all C files in that directory for the build without listing them individually.

- `target_sources(app PRIVATE ${app_sources})`

  Adding the source files specified in `app_sources` (which were located using the `FILE(GLOB ...)` command to the `app` target is the purpose of this command. The `app` target is the main executable target in Zephyr applications. Using the `PRIVATE` keyword indicates that these source files are exclusively used in the `app` target and not in any other target that might link against `app`.

main.c

The `main.c` file provided is a C program intended to operate on a Zephyr RTOS-based device. Its primary purpose is to configure a USB CDC ACM (Communications Device Class Abstract Control Model) device, which acts as a virtual serial port over USB and provides additional networking capabilities over USB. Below is an in-depth explanation of the file.

- **Includes and Logging Setup**

  The document contains different headers from the Zephyr RTOS, such as USB, UART, networking, and logging features. The logging is set up with the module name `cdc_acm_echo` and a logging level of `LOG_LEVEL_INF`.

- **Device and Ring Buffer Initialization**

  The CDC ACM interface's corresponding UART device can be obtained using the `DEVICE_DT_GET_ONE(zephyr_cdc_acm_uart)` macro. Initializing a ring buffer of 1024 bytes allows temporary storage of data sent or received over the UART interface. The `rx_throttled` flag handles flow control when the ring buffer reaches full capacity.

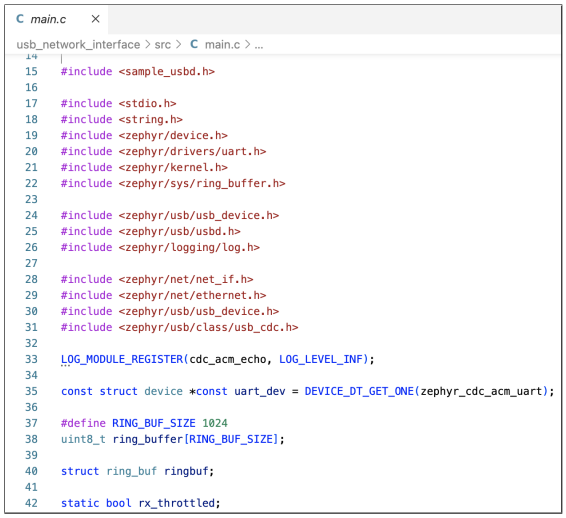


Figure 7.33: main.c part 1 - Setting Up the USB Network Interface

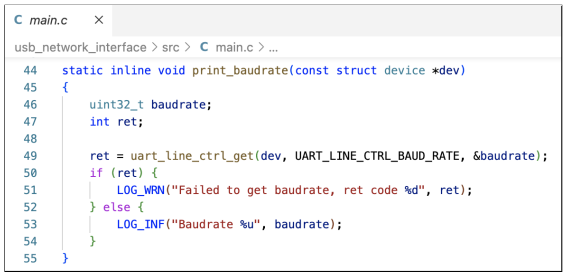


Figure 7.34: main.c part 2 - Setting Up the USB Network Interface

- **Baud Rate Printing Function** (`print_baudrate`)

  This process fetches and displays the UART device's current baud rate. It is employed when the line coding (baud rate) is altered.

- **USB Device Stack Initialization** (for `CONFIG_USB_DEVICE_STACK_NEXT`)

  When the newer USB device stack is enabled (`CONFIG_USB_DEVICE_STACK_NEXT`), USB device events like VBUS detection, control line state changes, and line coding changes are managed by the `sample_msg_cb` function. The `enable_usb_device_next` function is responsible for initializing and enabling the USB device.

- **Interrupt Handler**

Figure 7.35: main.c part 3 - Setting Up the USB Network Interface

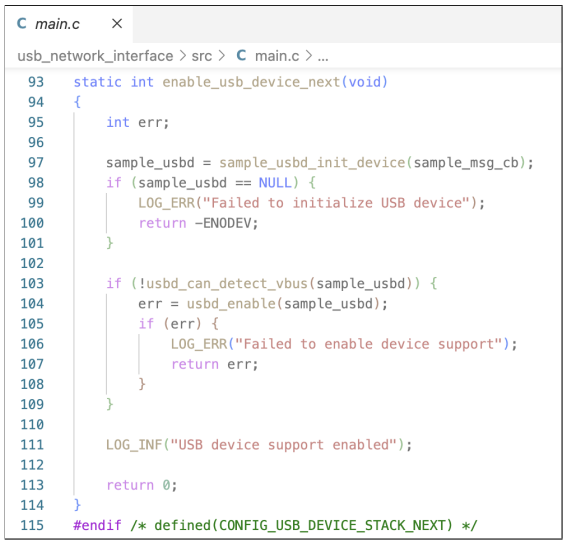


Figure 7.36: main.c part 4 - Setting Up the USB Network Interface

The function `interrupt_handler` manages UART interrupts for both sending and receiving data. Upon receiving data, it reads from the UART FIFO into a buffer and saves it in the ring buffer. When sending data, it transfers data from the ring buffer to the UART FIFO.

- **Main Function**

  The `main` function starts by verifying the readiness of the UART device (CDC ACM). Depending on whether the newer USB stack is being utilized, it either invokes `enable_usb_device_next()` or the legacy `usb_enable(NULL)` function to activate USB support. Subsequently, the program pauses to

main.c

usb_network_interface > src > main.c > ...

```c
static void interrupt_handler(const struct device *dev, void *user_data)
{
    ARG_UNUSED(user_data);

    while (uart_irq_update(dev) && uart_irq_is_pending(dev)) {
        if (!rx_throttled && uart_irq_rx_ready(dev)) {
            int recv_len, rb_len;
            uint8_t buffer[64];
            size_t len = MIN(ring_buf_space_get(&ringbuf),
                             sizeof(buffer));

            if (len == 0) {
                /* Throttle because ring buffer is full */
                uart_irq_rx_disable(dev);
                rx_throttled = true;
                continue;
            }

            recv_len = uart_fifo_read(dev, buffer, len);
            if (recv_len < 0) {
                LOG_ERR("Failed to read UART FIFO");
                recv_len = 0;
            };

            rb_len = ring_buf_put(&ringbuf, buffer, recv_len);
            if (rb_len < recv_len) {
                LOG_ERR("Drop %u bytes", recv_len - rb_len);
            }

            LOG_DBG("tty fifo -> ringbuf %d bytes", rb_len);
            if (rb_len) {
                uart_irq_tx_enable(dev);
            }
        }
```

Figure 7.37: main.c part 5 - Setting Up the USB Network Interface

receive the DTR (Data Terminal Ready) signal from the host, signifying the host's readiness to communicate with the device. Optionally, the DCD (Data Carrier Detect) and DSR (Data Set Ready) signals are configured, both standard serial control signals. After a brief pause, the baud rate is displayed (if the legacy USB stack is used), and the UART interrupt handler is configured. Finally, the RX interrupts are enabled, initiating the reception of data.

- **Network Interface Setup**

  The default network interface is obtained by using `net_if_get_default()`. The `net_if_up(iface)` function brings up the network interface, enabling the device to commence processing network traffic via the USB connection.

- **Purpose and Functionality**

  The software configures the device as a virtual serial port over USB, allowing for serial communication with a host computer. This capability is valuable for logging, debugging, or interacting with host applications. Along with its role

main.c

usb_network_interface > src > main.c > ...

```c
int main(void)
{
    int ret;

    if (!device_is_ready(uart_dev)) {
        LOG_ERR("CDC ACM device not ready");
        return 0;
    }

#if defined(CONFIG_USB_DEVICE_STACK_NEXT)
        ret = enable_usb_device_next();
#else
        ret = usb_enable(NULL);
#endif

    if (ret != 0) {
        LOG_ERR("Failed to enable USB");
        return 0;
    }

    ring_buf_init(&ringbuf, sizeof(ring_buffer), ring_buffer);

    LOG_INF("Wait for DTR");

#if defined(CONFIG_USB_DEVICE_STACK_NEXT)
    k_sem_take(&dtr_sem, K_FOREVER);
#else
    while (true) {
        uint32_t dtr = 0U;

        uart_line_ctrl_get(uart_dev, UART_LINE_CTRL_DTR, &dtr);
        if (dtr) {
            break;
        } else {
            /* Give CPU resources to low priority threads. */
            k_sleep(K_MSEC(100));
        }
    }
#endif
```

Figure 7.38: main.c part 6 - Setting Up the USB Network Interface

as a serial port, the software also sets up the device to operate as a network interface over USB. This functionality enables the device to manage network traffic and respond to API requests. Using UART interrupts ensures that the device can efficiently manage incoming and outgoing data without busy waiting, which is particularly essential in real-time systems such as those running on Zephyr RTOS. The code is designed to support both the traditional USB device stack and the newer `CONFIG_USB_DEVICE_STACK_NEXT`, making it adaptable across various versions or configurations of the Zephyr RTOS.

Output

The output from the serial terminal shows a USB CDC ACM (USB serial) device waiting for a Data Terminal Ready (DTR) signal before allowing additional communication. It also tries to establish a network interface over USB but has a

main.c

usb_network_interface > src > main.c > ...

```c
int main(void)
{
    LOG_INF("DTR set");

    /* They are optional, we use them to test the interrupt endpoint */
    ret = uart_line_ctrl_set(uart_dev, UART_LINE_CTRL_DCD, 1);
    if (ret) {
        LOG_WRN("Failed to set DCD, ret code %d", ret);
    }

    ret = uart_line_ctrl_set(uart_dev, UART_LINE_CTRL_DSR, 1);
    if (ret) {
        LOG_WRN("Failed to set DSR, ret code %d", ret);
    }

    /* Wait 100ms for the host to do all settings */
    k_msleep(100);

#ifndef CONFIG_USB_DEVICE_STACK_NEXT
    print_baudrate(uart_dev);
#endif
    uart_irq_callback_set(uart_dev, interrupt_handler);

    /* Enable rx interrupts */
    uart_irq_rx_enable(uart_dev);

    // Configure the network interface
    struct net_if *iface = net_if_get_default();
    if (!iface) {
        LOG_WRN("Failed to get default network interface");
        return;
    }

    //Set the network interface up
    net_if_up(iface);
    LOG_INF("USB network interface is up");

    return 0;
}
```

Figure 7.39: main.c part 7 - Setting Up the USB Network Interface

```
*** Booting nRF Connect SDK v3.5.99-ncs1-1 ***
[00:00:00.000,915] <inf> cdc_acm_echo: Wait for DTR


[00:00:05.004,333] <inf> cdc_acm_echo: DTR set
[00:00:05.119,842] <inf> cdc_acm_echo: Baudrate 115200
[00:00:05.120,025] <wrn> cdc_acm_echo: Failed to get default network interface
```

Figure 7.40: Output on Serial Terminal when running the USB Network Interface

problem. Here's an analysis of the output.

- **Boot Message**

  This shows that the device is starting up using the designated version of the nRF Connect SDK.

- **Waiting for DTR**

  Before proceeding, the device awaits the Data Terminal Ready (DTR) signal from the host, usually a PC or another USB host. DTR is a control signal utilized in serial communication to indicate the host's readiness to communicate.

- **DTR Set**

  The host has sent the DTR signal, indicating readiness for communication.

- **Baudrate Message**

  The device reads and logs the serial communication baud rate setting. It is currently configured to 115200 baud.

- **Network Interface Warning**

  The device's attempt to retrieve the default network interface was unsuccessful. This might be attributed to various factors, including misconfiguration, network interface unavailability, or a USB networking support problem.

Encountering the warning "Failed to Obtain the Default Network Interface" signifies that the network interface, which should be accessible through USB, is not correctly configured or initialized. This issue may result from missing or incorrectly configured networking components in the build configuration (`prj.conf`) or a USB-to-Ethernet implementation issue. Potential Solutions are Validate Configuration Ensure that the required network components (e.g., Ethernet over USB) are correctly configured in `prj.conf`. Verify USB and Network Support Confirm that the USB device stack and the network interface support are accurately implemented and initialized in the code.

Throughout our development process, we faced significant obstacles while setting up the USB network interface on the Nordic Thingy:53 using the nRF Connect SDK. We could not establish a stable network connection despite thorough troubleshooting and multiple configuration attempts, such as configuring the device as a USB CDC ACM (Communication Device Class Abstract Control Model). The main issue was the inability to properly initialize the default network interface, which hindered the device from obtaining an IP address and establishing a reliable connection. Due to time constraints, we decided to set this issue aside temporarily. We will explore alternative solutions or revisit this problem with a fresh perspective later. Our current focus will shift to other priorities.

Setting Up the USB HTTP Server and JSON API Endpoint

Source Code

Kconfig

The `Kconfig` file in the Zephyr project specifies and sets up different build options using a menu-based configuration system. The file you supplied is instrumental in configuring USB-related choices for a particular project. Here's an overview of what it contains.

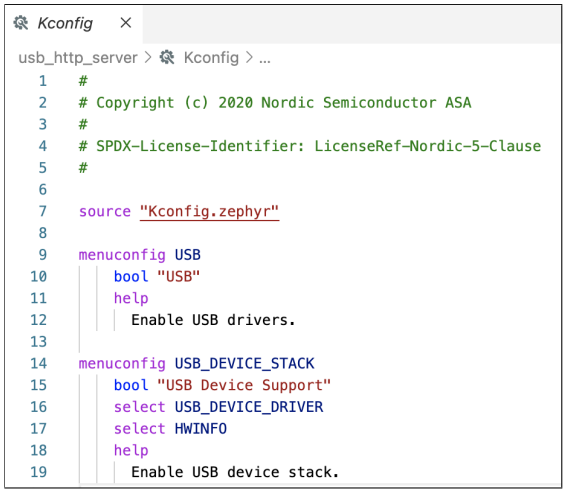


Figure 7.41: Kconfig - Setting Up the USB HTTP Server and JSON API Endpoint

- `source "Kconfig.zephyr"`

  This line imports the base `Kconfig` file from Zephyr, containing the core configuration options of the Zephyr RTOS.

- `menuconfig USB`

  Creates a menu configuration setting called `USB`. It is of type `bool` (boolean), allowing it to be turned on or off (true/false). This setting activates USB drivers in the project. The help section offers a short explanation of the purpose of this setting, specifically, activating USB drivers when it is selected.

- `menuconfig USB_DEVICE_STACK`

  Creates a menu configuration choice called `USB_DEVICE_STACK`. It is of type `bool` (boolean), which can be turned on or off. When `USB_DEVICE_STACK` is enabled, it automatically selects the `USB_DEVICE_DRIVER` option. This is probably necessary for the USB device stack to work properly. When `USB_DEVICE_STACK` is enabled, it also selects the `HWINFO` option. This option is typically used for accessing hardware information, such as unique device IDs, which may be essential for USB device functionality. Enabling this option allows the project to support the USB device stack, which is crucial for the device to function as a USB peripheral.

prj.conf

The file `prj.conf` is used to configure Zephyr-based projects. This file specifies various settings that dictate the behaviour of the Zephyr operating system, device drivers, networking, and application-specific functionalities. The configurations in this file establish a Zephyr-based project that utilizes USB CDC ACM for serial communication over USB, includes support for `C++11`, enables networking (IPv4 and DHCP), and hosts an HTTP server capable of serving JSON data. Additionally, the USB device stack and logging are activated, while more advanced features such as floating-point support and hardware stack protection are present but currently deactivated. Here's an overview of the key settings in the file.

- **General Kernel Configuration**

  The size of the main thread's stack is set to 4096 bytes using `CONFIG_MAIN_STACK_SIZE`. This is crucial to ensure the main thread has sufficient memory. The stack size for the system workqueue is defined as 2048 using `CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE`, which is utilized to process deferred work. The heap memory pool allocates 1024 bytes using `CONFIG_HEAP_MEM_POOL_SIZE` for dynamic memory allocation during runtime.

- **USB Device Stack Configuration**

prj.conf

usb_http_server > prj.conf

```
# General Kernel Configuration
CONFIG_MAIN_STACK_SIZE=4096
CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE=2048
CONFIG_HEAP_MEM_POOL_SIZE=1024

# Enable USB device stack
CONFIG_USB=y
CONFIG_USB_DEVICE_STACK=y
CONFIG_USB_DEVICE_INITIALIZE_AT_BOOT=y
CONFIG_USB_DEVICE_LOG_LEVEL_INF=y

# Enable USB Network support
#CONFIG_USB_DEVICE_NETWORK=y
CONFIG_USB_DEVICE_NETWORK_RNDIS=y

# enable c++ support
CONFIG_CPLUSPLUS=y
CONFIG_STD_CPP11=y
CONFIG_LIB_CPLUSPLUS=y
CONFIG_NEWLIB_LIBC=y
CONFIG_NEWLIB_LIBC_FLOAT_PRINTF=y

# enable floating point unit
#CONFIG_FPU=y
```

Figure 7.42: prj.conf part 1 - Setting Up the USB HTTP Server and JSON API Endpoint

Enabling USB support in the project is achieved by setting `CONFIG_USB=y`. Setting `CONFIG_USB_DEVICE_STACK=y` enables the USB device stack, allowing the device to operate as a USB device. To initialize the USB device stack at boot time, set `CONFIG_USB_DEVICE_INITIALIZE_AT_BOOT=y`. The logging level for the USB device stack can be configured to the "Info" level by setting `CONFIG_USB_DEVICE_LOG_LEVEL_INF=y`, which will result in the logging of informational messages.

- **USB CDC ACM (Communications Device Class - Abstract Control Model)**

  The option `CONFIG_USB_CDC_ACM` is set to `y`, which enables the USB CDC ACM class and allows the device to imitate a serial port over USB. The option `CONFIG_NET_L2_ETHERNET` is set to `y`, enabling Ethernet layer two networking support, allowing the device to utilize Ethernet for networking.

prj.conf

usb_http_server > prj.conf

```
26  # Enable Networking
27  CONFIG_NETWORKING=y
28  CONFIG_NET_IPV4=y
29  CONFIG_NET_DHCPV4=y
30  CONFIG_NET_TCP=y
31  CONFIG_NET_UDP=y
32  CONFIG_NET_SOCKETS=y
33  CONFIG_NET_SOCKETS_POSIX_NAMES=y
34  CONFIG_NET_SOCKETS_OFFLOAD=n
35  CONFIG_NET_CONFIG_AUTO_INIT=y
36  CONFIG_NET_CONFIG_SETTINGS=y
37  #CONFIG_NET_CONFIG_MY_IPV4_ADDR="192.168.2.2"
38  #CONFIG_NET_CONFIG_PEER_IPV4_ADDR="192.168.2.1"
39  #CONFIG_NET_CONFIG_MY_IPV4_NETMASK="255.255.255.0"
40  #CONFIG_NET_CONFIG_MY_IPV4_GW="192.168.2.1"
41
42
43  # Enable HTTP Server
44  CONFIG_HTTP_SERVER=y
45  #CONFIG_HTTP_SERVER_TIMEOUT=30000
46  CONFIG_HTTP_PARSER_URL=y
47
48  # Enable JSON support
49  CONFIG_JSON_LIBRARY=y
50  CONFIG_CJSON_LIB=y
51
52  # Logging
53  CONFIG_LOG=y
54  CONFIG_LOG_DEFAULT_LEVEL=4
55  #CONFIG_USB_DEVICE_LOG_LEVEL_DBG=y
56
57  # Enable stack protection
58  #CONFIG_HW_STACK_PROTECTION=y
```

Figure 7.43: prj.conf part 2 - Setting Up the USB HTTP Server and JSON API Endpoint

The option `CONFIG_ETH_NATIVE_POSIX` is set to `y`. This configuration typically supports POSIX-compliant Ethernet drivers, useful in native POSIX (Linux) environments.

- **C++ Support**

  `CONFIG_CPLUSPLUS=y` Activates `C++` language support in the project.
  `CONFIG_STD_CPP11=y` Allows for `C++11` standard support.
  `CONFIG_LIB_CPLUSPLUS=y` Adds the `C++` standard library. `CONFIG_NEWLIB_LIBC=y` Permits the usage of the Newlib `C` library, which is a lightweight `C` standard library implementation. `CONFIG_NEWLIB_LIBC_FLOAT_PRINTF=y` Enables support for floating-point functionality in printf functions within Newlib.

- **Floating Point Unit (FPU)**

  The `CONFIG_FPU=y` setting is commented out, suggesting that floating-point operations are not explicitly needed and are not specifically enabled.

- **Networking Configuration**

  Enable the networking stack in Zephyr with `CONFIG_NETWORKING=y`. IPv4 support for networking is enabled with `CONFIG_NET_IPV4=y`. The device can automatically obtain an IP address from a DHCP server with `CONFIG_NET_DHCPV4=y`. Support for TCP (Transmission Control Protocol) is enabled with `CONFIG_NET_TCP=y`. UDP (User Datagram Protocol) support is enabled with `CONFIG_NET_UDP=y`. Socket APIs for networking are enabled with `CONFIG_NET_SOCKETS=y`. POSIX-compliant socket names for ease of use are enabled with `CONFIG_NET_SOCKETS_POSIX_NAMES=y`. Socket offloading is disabled, meaning Zephyr's networking stack will handle all socket operations with `CONFIG_NET_SOCKETS_OFFLOAD=n`. Network configuration is automatically initialized at startup with `CONFIG_NET_CONFIG_AUTO_INIT=y`. Manual configuration of network settings is enabled with `CONFIG_NET_CONFIG_SETTINGS=y`, though the specific settings (IP address, netmask, gateway) are commented out.

- **HTTP Server Configuration**

  The built-in HTTP server in Zephyr is enabled by setting `CONFIG_HTTP_SERVER` to `y`. This allows the device to serve web pages or APIs. Enabling URL parsing in the HTTP server, which is necessary for handling different endpoints, is achieved by setting `CONFIG_HTTP_PARSER_URL` to `y`.

- **JSON Support**

  Enable the JSON library by setting `CONFIG_JSON_LIBRARY=y`. This will enable the device to work with JSON data. Additionally, by setting `CONFIG_CJSON_LIB=y`, you can include the `cJSON` library, a lightweight JSON parsing and formatting library.

- **Logging Configuration**

Enabling the logging system in Zephyr requires setting `CONFIG_LOG=y`. To establish the default logging level as "info" (level 4), which means that informational messages and higher levels (warnings, errors) will be recorded, set `CONFIG_LOG_DEFAULT_LEVEL=4`. The line responsible for debugging-level USB logs, `CONFIG_USB_DEVICE_LOG_LEVEL_DBG=y`, is currently commented out. As a result, only info-level logs will be generated unless this line is uncommented.

- **Stack Protection**

  The line `CONFIG_HW_STACK_PROTECTION=y` is currently commented out, indicating that hardware stack protection, which can help prevent stack overflows, is not activated.

CMakeLists.txt

The `CMakeLists.txt` file is used to configure the build process for a project called `usb_http_server` that is based on Zephyr and targets the application core of the Nordic Thingy:53. The file makes sure that it locates the necessary Zephyr environment, collects all C source files from the `src/` folder, and sets them up as the application's sources. This setup enables easy selection of the board and guarantees that all required files are compiled and linked correctly. Below is an explanation of the purpose of each part of the file.

- **CMake Version Requirement**

  This declaration sets the minimum version of CMake needed for building the project to be 3.20.0 or higher.

- **Board Configuration**

  The default board is designated as `thingy53_nrf5340_cpuapp` by this line. This board is the application core for the Nordic Thingy:53, a development board that utilizes the nRF5340 SoC (System on Chip). The provided comment signifies that an alternative board name can be specified to replace the default board by using the `-b` option when executing the build command.

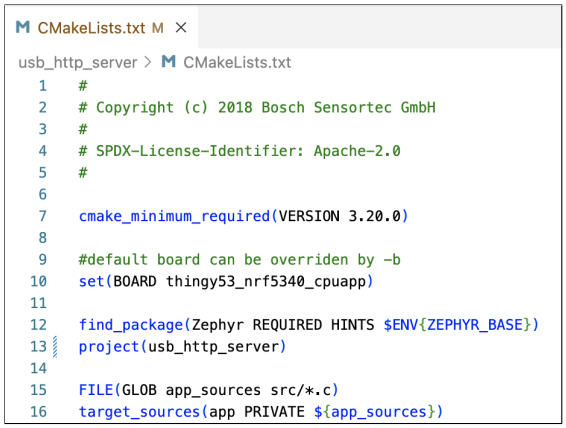


Figure 7.44: CMakeLists.txt - Setting Up the USB HTTP Server and JSON API Endpoint

- **Finding the Zephyr Package**

  This command instructs CMake to locate the Zephyr build system, which is essential for this project. The `"HINTS $ENV{ZEPHYR_BASE}"` portion implies that the Zephyr base directory should be identified in the `"ZEPHYR_BASE"` environment variable.

- **Project Declaration**

  The project name is specified as `usb_http_server` in this line. CMake uses this name to handle the build process and the resulting output files.

- **Source File Collection**

  The following statement gathers all the C source files (`*.c`) found in the `src/` folder and saves them in the `app_sources` variable. In this case, the `GLOB` command automatically includes all C source files without listing them individually.

- **Target Configuration**

  The following line identifies the source files that will be compiled for the target application (`app`). Using the `PRIVATE` keyword signifies that the `app_sources` are exclusively intended for this specific target and not accessible to others.

main.c

The `main.c` file contains the source code for an application based on Zephyr, which establishes an HTTP server on a device. This server is responsible for processing dynamic requests, fetching simulated sensor data, encoding it into JSON format, and sending it back as an HTTP response. Here is an in-depth explanation.

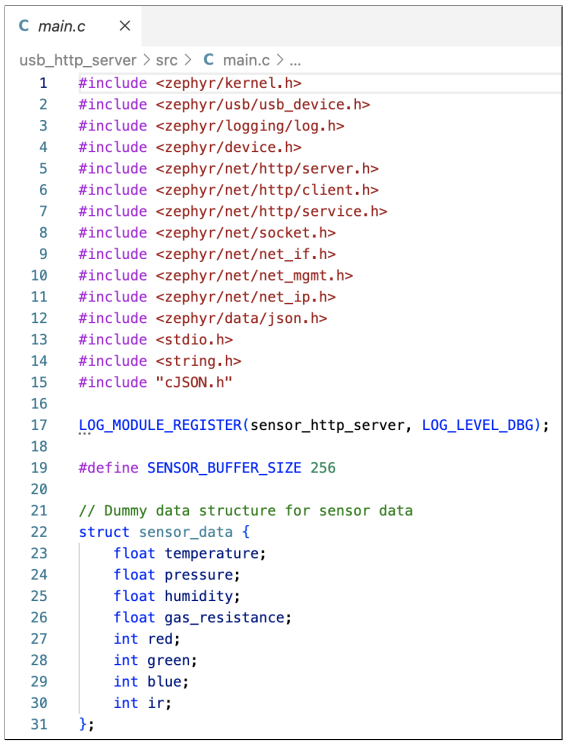


Figure 7.45: main.c part 1 - Setting Up the USB HTTP Server and JSON API Endpoint

- **Header Inclusions**

  The code begins by importing necessary Zephyr headers and libraries, such as `zephyr/kernel.h` for Kernel services and threads, `zephyr/usb/usb_device.h` for USB device handling, `zephyr/logging/log.h` for logging functionality, `zephyr/device.h` for the device model, `zephyr/net/http/server.h` for the HTTP server API, `zephyr/net/http/client.h` for the HTTP client API, `zephyr/net/http/service.h` for HTTP service handling, `zephyr/net/socket.h`

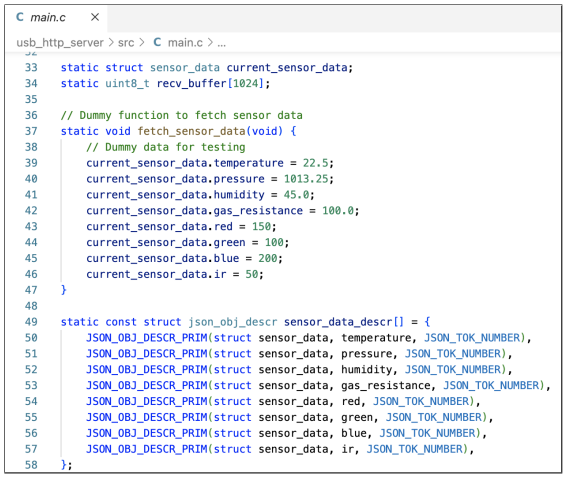


Figure 7.46: main.c part 2 - Setting Up the USB HTTP Server and JSON API Endpoint

for networking socket API, `zephyr/net/net_if.h` for network interface management, `zephyr/net/net_mgmt.h` for network management services, `zephyr/net/net_ip.h` for IP address management, `zephyr/data/json.h` for JSON encoding and decoding, standard C libraries such as `stdio.h` and `string.h`, as well as `cJSON.h` for JSON handling with cJSON library.

- **Module Logging**

  The logging module named `sensor_http_server` is registered with logging at the debug level.

- **Sensor Data Structure and Buffers**

  A structure, `sensor_data`, is defined to store dummy sensor data. A buffer (`recv_buffer`) is allocated to receive HTTP data.

- **Dummy Sensor Data Function**

  Populates `current_sensor_data` with placeholder values for testing purposes.

- **JSON Encoding Function**

  It retrieves sensor data and converts it into JSON format using Zephyr's JSON API.

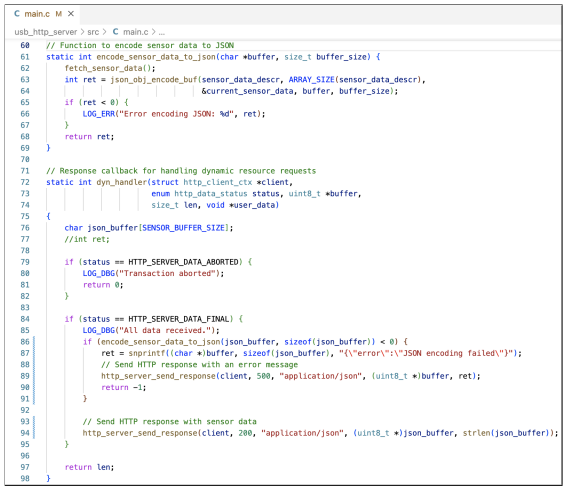


Figure 7.47: main.c part 3 - Setting Up the USB HTTP Server and JSON API Endpoint

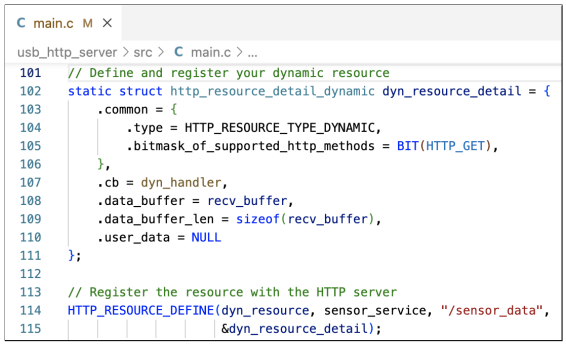


Figure 7.48: main.c part 4 - Setting Up the USB HTTP Server and JSON API Endpoint

- **HTTP Request Handling Function**

  When an HTTP request is received, it processes the incoming requests. If the request is fully received and processed (`HTTP_SERVER_DATA_FINAL`), it transforms the sensor data into JSON format and sends it back as the HTTP response. If the data cannot be encoded, it sends an error message.

- **HTTP Resource Definition**

  Creates a dynamic HTTP resource located at `/sensor_data`, managed by the `dyn_handler`.

main.c M

usb_http_server > src > main.c > ...

```c
void main(void) {
    int ret;

    // Initialize USB
    ret = usb_enable(NULL);
    if (ret != 0) {
        LOG_ERR("Failed to enable USB: %d", ret);
        return;
    }

    // Set up network interface
    struct net_if *iface = net_if_get_default();
    if (!iface) {
        LOG_ERR("No default network interface found");
        return;
    }

    // Start DHCP client to get IP address
    net_dhcpv4_start(iface);

    // Wait until IP is assigned (simplified wait)
    k_msleep(5000);

    // Start HTTP server
    ret = http_server_start();
    if (ret < 0) {
        LOG_ERR("Failed to start HTTP server: %d", ret);
        return;
    }

    LOG_INF("HTTP server started successfully");

    // Main loop
    while (1) {
        k_sleep(K_SECONDS(5)); // Simulate a periodic task
    }
}
```

Figure 7.49: main.c part 5 - Setting Up the USB HTTP Server and JSON API Endpoint

- **Main Function**

  USB Initialization: Activates USB functionality. Network Interface Setup: Retrieves the default network interface and initiates DHCP to obtain an IP address. HTTP Server Start: Initiates the HTTP server and goes into a main loop, emulating periodic tasks with `k_sleep()`.

Output

We faced many build errors when we tried to set up the USB HTTP server and JSON API endpoint on the Nordic Thingy:53. These problems varied from conflicts in configuration to unresolved dependencies within the nRF Connect SDK. Despite our attempts to troubleshoot and modify various settings, the build errors persisted, hindering us from establishing a functional setup. Due to time

Figure 7.50: Build Error for Setting Up the USB HTTP Server and JSON API Endpoint

constraints and the complexity of the issues, we have opted to postpone this task temporarily. We intend to explore alternative solutions or return to the problem with a fresh perspective. This approach will allow us to concentrate on other critical priorities while ensuring we can return to this challenge with a clearer understanding and a better-prepared strategy.

Exemplary Tools

nRF Connect for Desktop

nRF Connect for Desktop is a comprehensive software toolset designed to facilitate the development of Nordic Products across different platforms. nRF Connect for Desktop applications includes Bluetooth Low Energy, Board Configurator, Cellular Monitor, Direct Test Mode, nPM PowerUP, Power Profiler, Programmer, Quick Start, RSSI Viewer, Serial Terminal, and Toolchain Manager. The most commonly used apps are Programmer and Serial Terminal. Below are explanations about Programmer and Serial Terminal [18].

- **Programmer**

  The Programmer app has been developed to simplify Nordic Semiconductor's System on Chips (SoCs) programming. Users can quickly transfer files by dragging and dropping them into the app's interface. Furthermore, users can perform actions like reading, writing, or erasing data on the connected device. The app is focused on making the programming of Nordic SoCs easier, providing convenience and efficiency to its users. nRF Connect Programmer

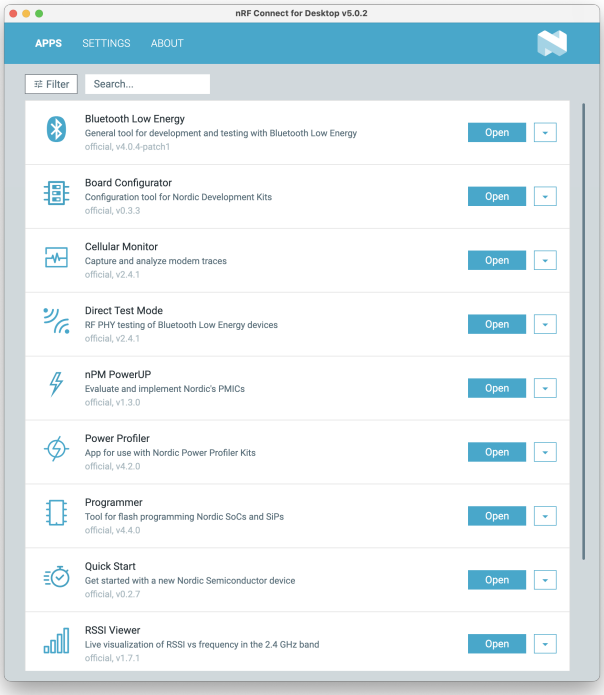


Figure 7.51: nRF Connect for Desktop

is an application offered by nRF Connect for Desktop, enabling users to program firmware to Nordic devices. This application provides visibility into the memory layout for both J-Link and Nordic USB devices, allowing users to view the content of HEX files and write them to the devices.

- **Serial Terminal**

  nRF Connect Serial Terminal is a terminal emulator used on different platforms to communicate with Nordic Semiconductor devices through a Universal Asynchronous Receiver/Transmitter (UART).

  The Serial Terminal lets you set up, monitor, and engage in virtual serial port communications with Nordic Semiconductor devices. It proves especially helpful for programming or debugging tasks, allowing you to observe logging outputs and input console commands. The terminal window supports shell mode for sending commands to devices running a shell-like Zephyr™ shell

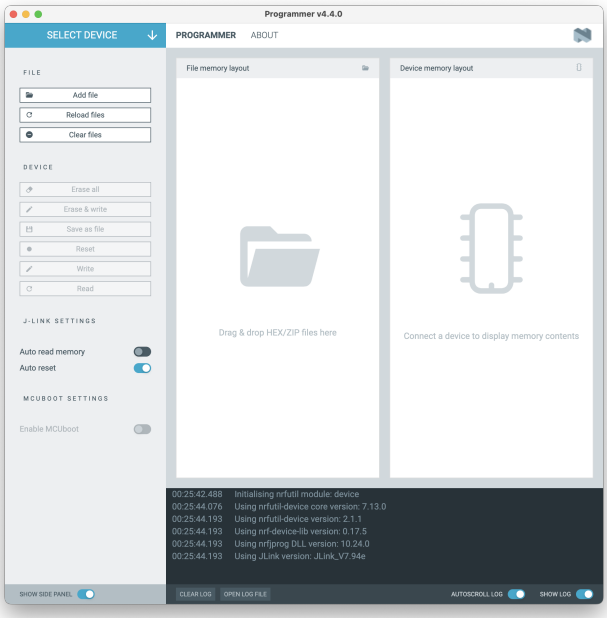


Figure 7.52: Programmer

and line mode. It also supports ANSI escape codes, making it simple to locate warnings and errors.

As an alternative to tools like PuTTY and minicom, Serial Terminal is tailored explicitly for Nordic Semiconductor devices. It features device autodetection, auto-reconnect, and persistence of device settings. Notably, terminal input and output are retained even after device disconnection. Serial Terminal facilitates shared access to a connected device's serial port with other compatible nRF Connect for Desktop applications.

Visual Studio Code (VS Code)

Visual Studio Code, a lightweight yet robust source code editor, is compatible with Windows, macOS, and Linux operating systems. It offers built-in support for JavaScript, TypeScript, and Node.js and boasts a diverse selection of extensions for other languages and runtimes, such as C++, C#, Java, Python, PHP, Go, and .NET [7].

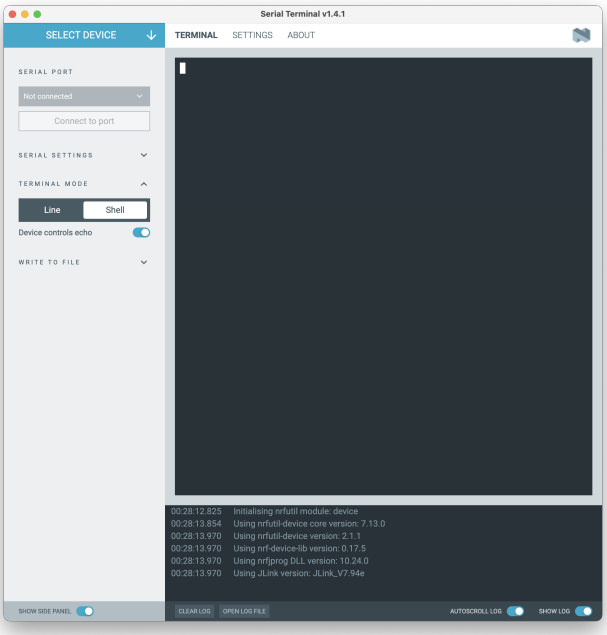


Figure 7.53: Serial Terminal

Another Experimentation

Reading Buttons and Controlling LEDs

The interaction between applications and hardware involves using software components known as device drivers. For our case study, we will examine the General-Purpose Input/Output (GPIO) hardware peripheral and its corresponding driver. We will review the blinky sample flashed onto the board line by line. In the exercise segment of this lesson, we will focus on learning how to manipulate LEDs and read button statuses using polling and interrupt methods with the GPIO peripheral.

This session will explore the Zephyr devicetree API, defined in the `<zephyr/devicetree.h>` header file. We will delve into board-level devicetree .dts files and SoC-level devicetree .dtsi files to understand how device configuration information is organized. We aim to comprehend the purpose of devicetree binding files (.yaml) and the compatible property.

Furthermore, we will focus on the device driver model, defined in `<zephyr/device.h>`, and examine the separation between a device driver API and its implementation.

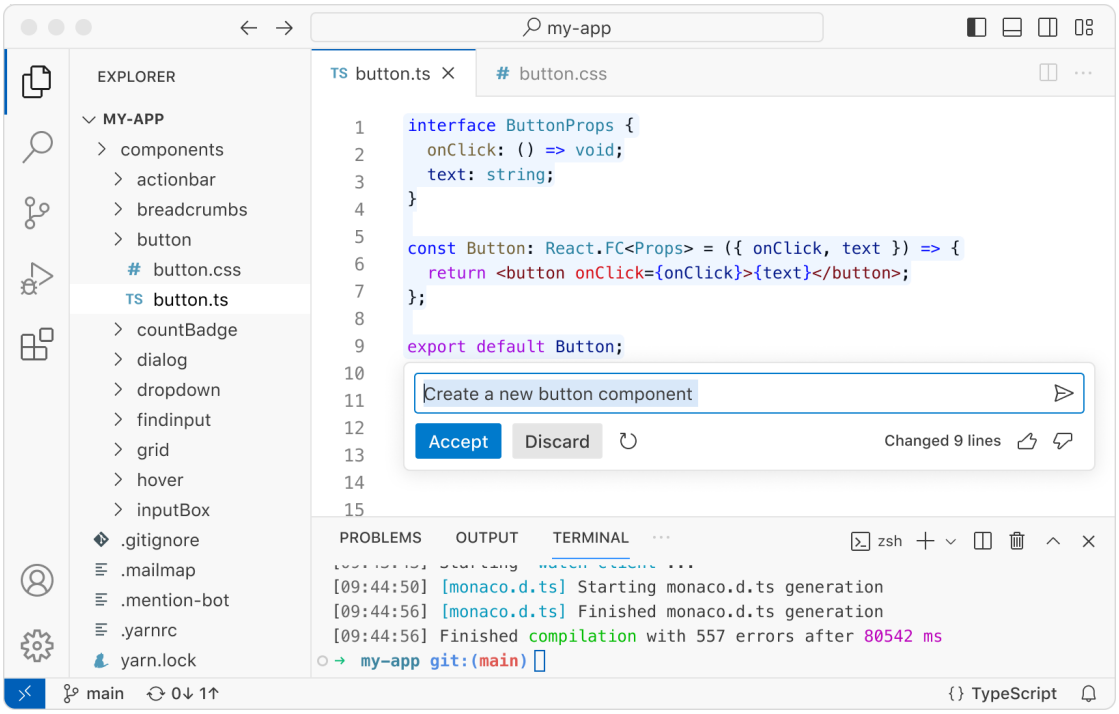


Figure 7.54: Visual Studio Code Interface

We will also discuss the significance of having a device pointer.

Additionally, we will analyze the generic GPIO interface APIs as defined in `<zephyr/drivers/gpio.h>` and engage in hands-on exercises to configure GPIO pins. This practical experience will involve reading from and writing to GPIO pins and setting up interrupts for input GPIO pins.

The `prj.conf` file within Zephyr projects serves as a configuration file that specifies the settings and features to be enabled or disabled in your firmware. It is essential to configure the build environment and manage the behaviour of your application and the Zephyr operating system.

`CONFIG_GPIO=y`

- `CONFIG_GPIO` is a setting in the Zephyr OS.
- This setting determines whether the General-Purpose Input/Output (GPIO) subsystem is turned on or off.

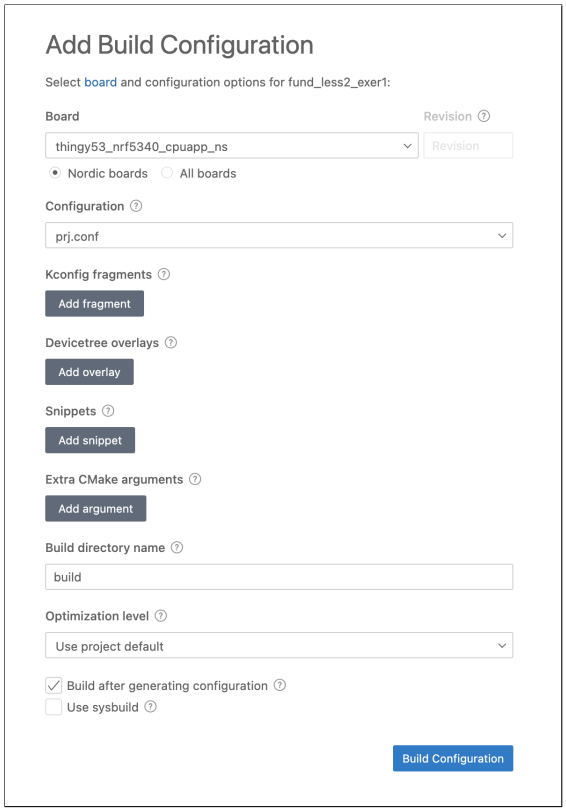


Figure 7.55: Add Build Configuration for Reading Buttons and Controlling LEDs

- When `CONFIG_GPIO` is set to y, it enables the GPIO subsystem in Zephyr, allowing your application to communicate with GPIO pins on the microcontroller.

General-Purpose Input/Output (GPIO) is an important interface that allows a microcontroller to communicate with external components such as sensors, LEDs, and buttons. GPIO pins can be set as inputs to read the status of buttons or sensors or as outputs to control the state of LEDs or other devices. Enabling the GPIO subsystem in your Zephyr project is crucial for embedded applications that require interaction with hardware peripherals. By enabling GPIO, your code gains the capability to both control and retrieve data from these pins. With GPIO enabled, your project can perform various tasks, including toggling LEDs, monitoring button states, and managing other hardware components connected to GPIO pins. This functionality is essential for embedded systems that require

prj.conf 6

prj.conf > ...

```
1  #
2  # Copyright (c) 2016 Intel Corporation
3  #
4  # SPDX-License-Identifier: Apache-2.0
5  #
6
7  CONFIG_GPIO=y
8
```

Figure 7.56: prj.conf for Reading Buttons and Controlling LEDs

precise control and communication with external devices.

main.c

src > main.c > main(void)

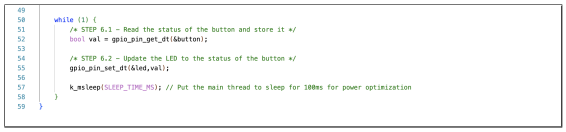
```c
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/devicetree.h>
#include <zephyr/drivers/gpio.h>

/* STEP 7 - Change the sleep time from 1000 ms to 100 ms */
#define SLEEP_TIME_MS   100

/* STEP 3.1 - Get the node identifier for button 1 through its alias sw0 */
#define SW0_NODE    DT_ALIAS(sw0)

/* STEP 3.2 - Get the device pointer, pin number, and pin's configuration flags through gpio_dt_spec */
static const struct gpio_dt_spec button = GPIO_DT_SPEC_GET(SW0_NODE, gpios);

/* LED0_NODE is the devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)
static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(LED0_NODE, gpios);


int main(void)
{
    int ret;

    if (!device_is_ready(led.port)) {
        return -1;
    }

    /* STEP 4 - Verify that the device is ready for use */
    if (!device_is_ready(button.port)) {
    return -1;
    }

    ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
    if (ret < 0) {
        return -1;
    }

    /* STEP 5 - Configure the pin connected to the button to be an input pin and set its hardware specifications */
    ret = gpio_pin_configure_dt(&button, GPIO_INPUT);
    if (ret < 0) {
        return -1;
    }
```

Figure 7.57: main.c part 1 - Reading Buttons and Controlling LEDs

The `main.c` file is the primary source file for a simple embedded application using the Zephyr RTOS. This code configures and controls an LED and a button on a microcontroller. It periodically checks the button's state and updates the LED accordingly.

1. **Include Necessary Headers**

   - `#include <zephyr/kernel.h>`:



```
49
50      while (1) {
51          /* STEP 6.1 - Read the status of the button and store it */
52          bool val = gpio_pin_get_dt(&button);
53
54          /* STEP 6.2 - Update the LED to the status of the button */
55          gpio_pin_set_dt(&led,val);
56
57          k_msleep(SLEEP_TIME_MS); // Put the main thread to sleep for 100ms for power optimization
58      }
59  }
```

Figure 7.58: main.c part 2 - Reading Buttons and Controlling LEDs

Including `<zephyr/kernel.h>` header provides access to Zephyr's kernel APIs, including functions for thread management, synchronization, timing, and power management (e.g., `k_msleep`).

- `#include <zephyr/device.h>`:

  Including `<zephyr/device.h>` header provides APIs for interacting with device drivers, enabling checking device readiness and retrieving device pointers.

- `#include <zephyr/devicetree.h>`:

  Including `<zephyr/devicetree.h>` header makes interaction with the device tree, a data structure describing hardware configuration, possible. This facilitates obtaining node identifiers, pin configurations, and other hardware-related settings.

- `#include <zephyr/drivers/gpio.h>`:

  With the inclusion of `<zephyr/drivers/gpio.h>` header, access to the GPIO (General-Purpose Input/Output) driver APIs is granted. These APIs are utilized to configure GPIO pins, read input states, and set output states.

2. **Define Constants and Macros**

   - `SLEEP_TIME_MS`:

     This parameter sets the duration of the sleep in milliseconds. The main loop will wait for this duration during each iteration. In STEP 7, it is suggested that this value be modified from 1000 ms (1 second) to 100 ms, which will cause the loop to check the button state ten times more often.

3. **Define GPIO Node Identifiers and Specifications**

- **Button Configuration**
  - `SW0_NODE`:
    This macro obtains the button's node identifier (usually the `sw0` alias) from the device tree. The device tree is a file that describes the hardware and its connections.
  - `gpio_dt_spec`:
    This structure holds the GPIO device pointer, pin number, and pin configuration flags. `GPIO_DT_SPEC_GET` is a macro that fetches this information from the device tree node (`SW0_NODE`).
- **LED Configuration**
  - `LED0_NODE`:
    Like `SW0_NODE`, this macro retrieves the node identifier for the LED (usually the `led0` alias) from the device tree.
  - `led`:
    It is a `gpio_dt_spec` structure for the LED, which includes the GPIO device pointer, the pin number, and the configuration flags.

4. **Main Function**

- `device_is_ready`:
  This function verifies whether the device (in this instance, the GPIO port for the LED and button) is prepared for use. If it is not, the program will return -1, indicating an error.

5. **Configure GPIO Pins**

- `gpio_pin_configure_dt`:
  This function sets up a GPIO pin using the `gpio_dt_spec` structure.
  To control the LED, the pin is set up as an output (`GPIO_OUTPUT_ACTIVE`), enabling it to turn the LED on or off by driving it high or low.
- **Button Setup:**
  The button pin is set up as an input (`GPIO_INPUT`). This allows the program to check the button's state (pressed or not pressed).

6. **Main Loop**

   - `gpio_pin_get_dt`:

     This function retrieves the current status of the button pin. It returns a value of `true (1)` if the button is pressed and `false (0)` if it is not pressed.

   - `gpio_pin_set_dt`:

     This function adjusts the state of the LED pin based on the value obtained from the button. If the button is pressed, the LED is switched on (`val` is true). The LED is switched off if the button is not pressed (`val` is false).

   - `k_msleep`:

     This function suspends the main thread for the specified duration (`SLEEP_TIME_MS`, which is 100 ms). The sleep interval aids in reducing CPU usage and power consumption, thereby enhancing the efficiency of the application.

The code aims to create a basic embedded application on a Zephyr-based system. The process involves the following steps:

1. **Initialization:** The code verifies the readiness of the devices (LED and button).

2. **Configuration:** It sets up the LED as an output and the button as an input.

3. **Loop:** It continuously monitors the button state and updates the LED to match its state, with a brief pause between each check.

This arrangement allows the microcontroller to switch an LED on or off based on the button press, which is a straightforward but common task in embedded systems.

The `CMakeLists.txt` file is a script utilized by `CMake`, a build system generator, to specify a project's build requirements, including the minimum required `CMake` version, package discovery, and build target and source file definitions.

CMakeLists.txt

```
#
# Copyright (c) 2016 Intel Corporation
#
# SPDX-License-Identifier: Apache-2.0
#

cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(nrf_connect_sdk_fundamentals)

target_sources(app PRIVATE src/main.c)

```

Figure 7.59: CMakeLists.txt for Reading Buttons and Controlling LEDs

- `cmake_minimum_required(VERSION 3.20.0)`

  The line sets the minimum `CMake` version needed for the project to build. Specifically, it requires CMake version 3.20.0 or higher. If the system has an older version, the build process will terminate with an error indicating the version mismatch.

- `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})`

  instructs `CMake` to find the Zephyr project package essential for the build. The `REQUIRED` keyword means that failure to locate Zephyr will halter the build process with an error. The `HINTS` keyword guides `CMake` where to search for Zephyr, utilizing the `ZEPHYR_BASE` environment variable to locate the Zephyr SDK. Typically, this environment variable is configured during the installation and setup of the Zephyr development environment.

- `project(nrf_connect_sdk_fundamentals)`

  The line establishes the project's name, `nrf_connect_sdk_fundamentals`, as an identifier for the project, utilized by `CMake` for internal project management as well as in generated build files and output directories.

- `target_sources(app PRIVATE src/main.c)`

  Specifies the source files for the build target.

- `target_sources(app ...` denotes the addition of source files to the `app` target, representing the application being constructed.
- `PRIVATE` keyword defines the scope of the added source files, indicating that they are exclusively used for building the specific target (`app`) and are not exposed to or reused by other targets.
- `src/main.c` denotes the path to the source file within the project. In this case, `main.c` is in the `src` directory. The application's main source file contains the program's entry point and core logic.

This `CMakeLists.txt` file configures a fundamental Zephyr project, ensuring the use of `CMake 3.20.0` or later, identifying the Zephyr SDK location via the `ZEPHYR_BASE` environment variable, specifying the project name as `nrf_connect_sdk_fundamentals`, and designating `main.c` in the `src` directory as the main source file for the application. This configuration enables `CMake` to compile `main.c` into an executable using the Zephyr SDK, linking it with essential Zephyr components and libraries for the build process.

Elements of an nRF Connect SDK application

In the nRF Connect SDK, an application comprises various elements used by the build system to generate the final runnable file. Comprehending these elements, their purposes, and their interactions is essential when developing your application. This activity will involve an in-depth exploration of each element to grasp their interrelationships. During this exercise, we will establish a basic functioning application from the beginning and integrate custom files and configurations to personalize the application.

As part of this training, we aim to achieve the following objectives. First, we want to gain a comprehensive understanding of the use of Kconfig configuration files, which enable and configure various software modules available in the nRF Connect SDK. Additionally, we aim to closely examine an application configuration file and a board configuration file to understand their relationship. Furthermore, we will learn how to explore the available configuration options of a certain software module using guiconfig. We will also delve into multi-image builds

and understand how a child's image is added to an application. Finally, through hands-on exercises, we will practice creating an application from scratch, adding modules using Kconfig, and modifying the devicetree to tailor the application to our specific requirements.

Printing Messages to Console and Logging

In a new software development environment, learning how to display messages, such as the famous "Hello World!" on a console, is crucial. Earlier, we briefly covered using printk() to output basic messages on the console. This time, we'll delve into logging further, exploring the basic approach with printk() and a more advanced method using the logging module. We will start by practising the easy-to-use printk() function to display string literals and formatted strings on a console. Then, we will set up/configure the feature-rich logger module to display string literals, formatted strings, and hexadecimal dump variables. Finally, we will look closer at the logger module's features.

In this module, we aim to learn how to print and format strings to a console using the printk() function. This includes understanding the syntax and usage of printk() for displaying strings and formatted output to the console. We will also learn to recognize the limitations of printk() and understand when it may not be suitable for certain debugging or logging scenarios.

Learning to print and format strings for a console using the logger module. This will involve understanding the benefits and additional features the logger module offers for efficient debugging and logging. Learn how to effectively hex dump variables using the logger module, enhancing the capability to analyze and debug data in hexadecimal format. Explore the logger module's various features and functionalities to understand its system debugging and analysis capabilities comprehensively. Engage in hands-on exercises to practice enabling and configuring software modules, allowing practical application of the concepts and techniques learned throughout the module. These exercises will provide the opportunity to implement the knowledge gained and build proficiency in utilizing the software modules for effective debugging and logging.

main.c

fund_less3_exer1 > src > main.c

```c
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>

int main(void)
{
    while(1) {
        printk("Hello World!\n\r");
        k_msleep(1000);
    }
}
```

Figure 7.60: main.c for Printing Messages to Console and Logging

The `main.c` file is a basic program created for the Zephyr RTOS (Real-Time Operating System). It illustrates a simple application that repeatedly prints `"Hello World!"` every second. Below is an explanation of the code:

- `#include <zephyr/kernel.h>`

  This line imports the Zephyr kernel header file. The kernel header provides access to the essential functionality of the Zephyr RTOS, including task management, synchronization, timers, and other core kernel features. This inclusion enables functions such as `k_msleep` to delay execution in this program.

- `#include <zephyr/sys/printk.h>`

  This line imports the header file for Zephyr's `printk` function, a streamlined output function for printing text to the console or debug output. It's akin to the standard `C printf` function but optimized for embedded systems with limited resources.

- `int main(void)`

  This line defines the `main` function as the program's entry point. Execution begins at this function when the program starts.

- `while(1) { ... }`

The `while(1)` loop is an endless loop. The code inside this loop runs repetitively until the system is reset or powered off. This is a common practice in embedded programming, where the system continually performs its task.

- `printk("Hello World!\n\r");`

  This line utilizes the `printk` function to display the text `"Hello World!"` followed by a newline (`\n`) and a carriage return (`\r`). The newline character moves the cursor to the next line, and the carriage return moves it to the beginning of the line. Together, they ensure that the next printed text starts on a new line at the beginning.

- `k_msleep(1000);`

  This line invokes the `k_msleep` function, which is a Zephyr kernel function that suspends the execution of the current thread for a specified number of milliseconds. In this case, the delay is 1000 milliseconds or 1 second. This function effectively pauses the loop for 1 second before the next iteration, causing the `"Hello World!"` message to be printed once per second.

The program starts by executing the `main` function, beginning the initialization process. Following initialization, the program enters an infinite loop denoted by `while(1)`, ensuring it continuously carries out its designated task. Within this loop, the program utilizes the `printk` function to output the message `"Hello World!"` to the console or debug output. Subsequently, the program pauses for 1 second using the `k_msleep(1000)` function before repeating the entire process. This straightforward program is a common introductory example for embedded systems, often used to verify the development environment's correct setup and ensure that basic functionality is operating as expected.

The `CMakeLists.txt` file is a configuration script used by CMake, a build system generator. It plays a crucial role in defining the build process for applications within the Zephyr RTOS environment. Below is a comprehensive breakdown of every line within the `CMakeLists.txt` file:

- `cmake_minimum_required(VERSION 3.20.0)`

CMakeLists.txt

fund_less3_exer1 > CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.20.0)
2  find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
3  project(hello_world)
4  target_sources(app PRIVATE src/main.c)
```

Figure 7.61: CMakeLists.txt for Printing Messages to Console and Logging

This line sets the minimum CMake version required to build the project. The script relies on CMake version 3.20.0 or higher, and attempting to build the project with an older CMake version will prompt an error message requesting an upgrade.

- `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})`

  instructs `CMake` to find the Zephyr project package essential for the build. The `REQUIRED` keyword means that failure to locate Zephyr will result in halting the build process with an error. The `HINTS` keyword guides `CMake` on where to search for Zephyr, utilizing the `ZEPHYR_BASE` environment variable to locate the Zephyr SDK. Typically, this environment variable is configured during the installation and setup of the Zephyr development environment.

- `project(hello_world)`

  This line declares the project's name, in this case, `hello_world`. Additionally, the `project` command initializes specific variables `CMake` uses for project building, such as the project name, version, and languages (such as C or C++). In the context of Zephyr applications, the project name commonly serves to identify the application within the build system.

- `target_sources(app PRIVATE src/main.c)`

  Specifies the source files for the build target.

  - `target_sources(app ...` denotes the addition of source files to the `app` target, representing the application being constructed.

- `PRIVATE` keyword defines the scope of the added source files, indicating that they are exclusively used for building the specific target (`app`) and are not exposed to or reused by other targets.
- `src/main.c` denotes the path to the source file within the project. In this case, `main.c` is in the `src` directory. The application's main source file contains the program's entry point and core logic.

This `CMakeLists.txt` file is typical for a straightforward Zephyr project. It establishes the essential environment for building a Zephyr application, ensuring the required tools and source files are appropriately configured. CMake generates the necessary build files to compile and link the `main after processing this file.c` file into an executable capable of running on a target device compatible with Zephyr.

Serial Communication (I2C)

Microcontrollers frequently work with sensors and external components. The Inter-Integrated Circuit (I2C) bus is a widely used protocol for communicating with external components. This tutorial will cover the fundamentals of I2C and understand how to utilize the I2C driver in the nRF Connect SDK to connect with an external sensor.

The configuration file `prj.conf` is utilized in projects based on Zephyr to set up different aspects and modules of the Zephyr RTOS. It is an essential text document where every line signifies a specific configuration choice, generally activating or deactivating particular features or defining specific parameters for the application. Below is an in-depth explanation of the `prj.conf` file:

- `CONFIG_I2C=y`

  The line enables the Zephyr project's I2C (Inter-Integrated Circuit) driver. When you set `CONFIG_I2C=y`, you enable the I2C driver, allowing the application to communicate with I2C-compatible devices such as sensors and displays. Once enabled, your application can use Zephyr's I2C APIs to interact with devices on the I2C bus.

prj.conf 1 ×

fund_less6_exer2 > prj.conf

```
#
# Copyright (c) 2016 Intel Corporation
#
# SPDX-License-Identifier: Apache-2.0
#

# STEP 2 - Enable the I2C driver
CONFIG_I2C=y

# disable logging for TF-M
CONFIG_TFM_SECURE_UART=n
CONFIG_TFM_LOG_LEVEL_SILENCE=y

# STEP 4.2 - Enable floating point format specifiers
CONFIG_CBPRINTF_FP_SUPPORT=y
CONFIG_SENSOR=y
```

Figure 7.62: prj.conf for Connecting to the BH1749 Ambient Light Sensor on Thingy:53

- `CONFIG_TFM_SECURE_UART=n`

  This line disables the Secure UART in the TF-M (Trusted Firmware-M) module. Trusted Firmware-M (TF-M) is an open-source reference implementation of secure world software for ARM Cortex-M processors. It provides secure services like secure storage, cryptography, and secure boot. In secure environments, a Secure UART, which stands for Universal Asynchronous Receiver/Transmitter, might be used for secure communication. Disabling this feature by setting `CONFIG_TFM_SECURE_UART=n` means that TF-M will not handle UART communication in a secure context. This might be useful if secure UART is unnecessary or a non-secure context is preferred.

- `CONFIG_TFM_LOG_LEVEL_SILENCE=y`

  Setting `CONFIG_TFM_LOG_LEVEL_SILENCE=y` is to silence logging in the TF-M module. This configuration option controls the verbosity of logs generated by TF-M, with available log levels typically including `SILENCE`, `ERROR`, `WARNING`, `INFO`, and `DEBUG`. By enabling `CONFIG_TFM_LOG_LEVEL_SILENCE=y`, all logging output from the TF-M module is suppressed. This may be done to reduce the amount of output, save memory, or prevent potential leakage of sensitive information in a secure environment.

- `CONFIG_CBPRINTF_FP_SUPPORT=y`

  In Zephyr, the line `CONFIG_CBPRINTF_FP_SUPPORT=y` supports floating-point format specifiers in the `cbprintf` function. The `cbprintf` function is used for formatted output in Zephyr, similar to `printf`, and is optimized for constrained environments to allow efficient printing. Enabling `CONFIG_CBPRINTF_FP_SUPPORT=y` allows the `cbprintf` function to handle floating-point numbers (`%f` specifier) in its formatted output. This is crucial when your application needs to display or log floating-point data, such as sensor readings.

- `CONFIG_SENSOR=y`

  The following line enables the sensor subsystem in Zephyr. This subsystem provides a unified API for interacting with various sensors, abstracting the details of sensor communication and allowing for easier integration and management of different types of sensors. By setting `CONFIG_SENSOR=y`, you enable the sensor framework in Zephyr, which enables your application to interact with supported sensors using the Zephyr sensor APIs.

The `prj.conf` file configures a Zephyr project to enable the I2C driver for communication with I2C devices, access the Zephyr sensor APIs for interacting with various sensors in the sensor subsystem, support the use of floating-point specifiers in formatted output, and disable secure UART and silence logging in TF-M, likely for a non-secure or streamlined operation.

The file `thingy53_nrf5340_cpuapp_ns.overlay` serves as a DeviceTree overlay designed to configure hardware peripherals on the Nordic Thingy:53 device, with a focus on the non-secure application core (`cpuapp_ns`) of the nRF5340 SoC (System on Chip). This overlay file is specifically tailored to customize the configuration of I2C peripherals on the board, particularly for setting up the BH1749 sensor. The label `\&i2c1` is used to identify the I2C1 controller on the device. Let's now delve into a detailed analysis of the code.

- `&i2c1{ ... }`

  `&i2c1` is about the I2C1 controller node, previously specified in the main DeviceTree for the board. The `&` symbol is utilized to refer to this preexisting

thingy53_nrf5340_cpuapp_ns.overlay

fund_less6_exer2 > boards > thingy53_nrf5340_cpuapp_ns.overlay

```
/*
* Copyright (c) 2021 Nordic Semiconductor ASA
*
* SPDX-License-Identifier: LicenseRef-Nordic-5-Clause
*/

&i2c1{
    bh1749: bh1749@38 {
            compatible = "rohm,bh1749";
            reg = <0x38>;
            int-gpios = <&gpio1 5 (GPIO_PULL_UP |GPIO_ACTIVE_LOW)>;
    };
};
```

Figure 7.63: overlay for Connecting to the BH1749 Ambient Light Sensor on Thingy:53

node. The content enclosed within `{ ... }` customizes or supplements extra properties to the `i2c1` controller.

- `bh1749: bh1749@38 { ... }`

  The BH1749 sensor node is labelled as `bh1749` in the DeviceTree. These labels are convenient for referring to the node in other parts of the code, such as in drivers. `bh1749@38` is denotes the BH1749 sensor that is situated at the I2C address `0x38`. The `@38` indicates that the I2C address of the sensor is `0x38`. This is how the I2C controller recognizes the BH1749 sensor on the bus.

- `compatible = "rohm,bh1749";`

  This property defines the compatible string that identifies the sensor with the BH1749. The string `"rohm,bh1749"` in the `compatible` field signifies that the device is suitable for the driver for the BH1749 sensor, which is a light sensor produced by Rohm Semiconductor.

- `reg = <0x38>;`

  `reg` is the device's address on the I2C bus as defined by this property. The I2C address of the BH1749 sensor is `<0x38>`. This is the location on the bus where the I2C controller will communicate with the sensor.

- `int-gpios = <\&gpio1 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;`

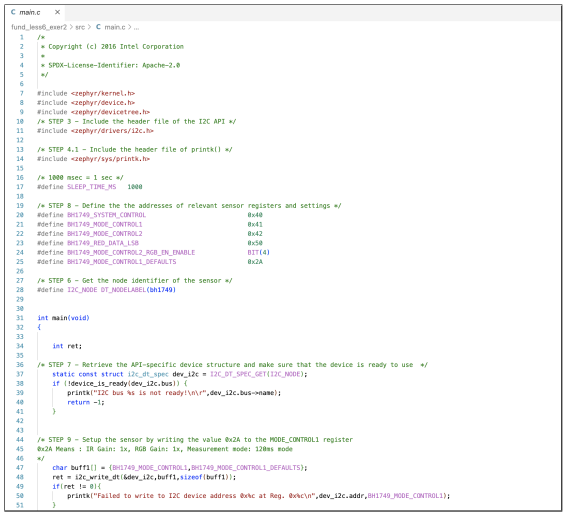
  The `int-gpios` property specifies the GPIO (General-Purpose Input/Output) pin utilized for the sensor's interrupt line. The `<\&gpio1 5 ...>` designation points to the GPIO controller and the pin number utilized for the interrupt. The `\&gpio1` references GPIO controller 1 (`gpio1`) on the SoC; this node is defined elsewhere in the DeviceTree. The number `5` denotes the pin number on the GPIO controller connected to the interrupt pin of the BH1749 sensor. Additionally, the settings `GPIO_PULL_UP` and `GPIO_ACTIVE_LOW` configure the GPIO pin with a pull-up resistor, causing it to default to a high state when not driven and indicate that the interrupt is active low, signifying that the sensor pulls the interrupt line low to signal an interrupt.

The overlay sets up the I2C1 bus on the Nordic Thingy:53 to communicate with a light sensor identified as BH1749 at address `0x38`. The sensor is linked through the I2C bus and utilizes GPIO1 pin 5 for an interrupt line configured with a pull-up resistor and active low. When this overlay is active, the Zephyr application can communicate with the BH1749 sensor using the appropriate drivers, managing interrupts from the sensor on the designated GPIO pin. This configuration is crucial for enabling the Zephyr RTOS to properly control and connect with the BH1749 sensor hardware, allowing the retrieval of light-intensity data and response to sensor-triggered interrupts.

The C program in the `main.c` file is intended to operate on a Zephyr RTOS-based platform. It communicates with the BH1749 light sensor via the I2C bus. This sensor can detect red, green, and blue light levels, and the program retrieves these values and displays them on the console periodically. Here is a comprehensive breakdown of the code:

- **Includes and Defines**

  In the codebase, several header files provide essential functionalities. For instance, `kernel.h` offers important functionalities like sleep and threading. Additionally, the `device.h` and `devicetree.h` files are utilized for accessing devices and nodes defined in the device tree. The `i2c.h` header file supplies the API for interacting with I2C devices, while `printk.h` includes the `printk()` function for recording output to the console.



```c
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/devicetree.h>
/* STEP 3 - Include the header file of the I2C API */
#include <zephyr/drivers/i2c.h>

/* STEP 4.1 - Include the header file of printk() */
#include <zephyr/sys/printk.h>

/* 1000 msec = 1 sec */
#define SLEEP_TIME_MS   1000

/* STEP 8 - Define the the addresses of relevant sensor registers and settings */
#define BH1749_SYSTEM_CONTROL                   0x40
#define BH1749_MODE_CONTROL1                    0x41
#define BH1749_MODE_CONTROL2                    0x42
#define BH1749_RED_DATA_LSB                     0x50
#define BH1749_MODE_CONTROL2_RGB_EN_ENABLE      BIT(4)
#define BH1749_MODE_CONTROL1_DEFAULTS           0x2A

/* STEP 6 - Get the node identifier of the sensor */
#define I2C_NODE DT_NODELABEL(bh1749)


int main(void)
{

    int ret;

/* STEP 7 - Retrieve the API-specific device structure and make sure that the device is ready to use  */
    static const struct i2c_dt_spec dev_i2c = I2C_DT_SPEC_GET(I2C_NODE);
    if (!device_is_ready(dev_i2c.bus)) {
        printk("I2C bus %s is not ready!\n\r",dev_i2c.bus->name);
        return -1;
    }


/* STEP 9 - Setup the sensor by writing the value 0x2A to the MODE_CONTROL1 register
0x2A Means : IR Gain: 1x, RGB Gain: 1x, Measurement mode: 120ms mode
*/
    char buff1[] = {BH1749_MODE_CONTROL1,BH1749_MODE_CONTROL1_DEFAULTS};
    ret = i2c_write_dt(&dev_i2c,buff1,sizeof(buff1));
    if(ret != 0){
        printk("Failed to write to I2C device address 0x%x at Reg. 0x%x\n",dev_i2c.addr,BH1749_MODE_CONTROL1);
    }
```

Figure 7.64: main.c part 1 - Connecting to the BH1749

```c

/* STEP 10 - Enable measurement by writing 1 to bit 4 of the MODE_CONTROL2 register */
    char buff2[] = {BH1749_MODE_CONTROL2,BH1749_MODE_CONTROL2_RGB_EN_ENABLE};
    ret = i2c_write_dt(&dev_i2c,buff2,sizeof(buff2));
    if(ret != 0){
        printk("Failed to write to I2C device address 0x%x at Reg. 0x%x\n",dev_i2c.addr,BH1749_MODE_CONTROL2);
    }

    while (1) {
/* STEP 11 - Read the RGB values from the sensor */
        uint8_t rgb_value[8]= {0};
        //Do a burst read of 6 bytes as each color channel is 2 bytes
        ret = i2c_burst_read_dt(&dev_i2c, BH1749_RED_DATA_LSB,rgb_value,sizeof(rgb_value));
        if(ret != 0){
            printk("Failed to read to I2C device address 0x%x at Reg. 0x%x\n",dev_i2c.addr,BH1749_RED_DATA_LSB);
        }
        //Print reading to console
        printk("_____________________________\n");
        printk("Red Value:\t %d\n", (rgb_value[0] | rgb_value[1] << 8));
        printk("Green Value:\t %d\n", (rgb_value[2] | rgb_value[3] << 8));
        printk("Blue Value:\t %d\n", (rgb_value[4] | rgb_value[5] << 8));

        k_msleep(SLEEP_TIME_MS);
    }
}
```

Figure 7.65: main.c part 2 - Connecting to the BH1749

Moreover, within the code, the `SLEEP_TIME_MS` constant is used to establish the sleep time between reading the sensor data, which is currently set to 1000 milliseconds (1 second). Various other definitions specify the addresses of registers and control bits specific to the BH1749 sensor. These definitions are employed to configure the sensor and read its data.

- **Initialization**

  `I2C_NODE` retrieves the I2C node for the BH1749 sensor from the device tree using the `DT_NODELABEL` macro. The `dev_i2c` structure is filled using the `I2C_DT_SPEC_GET` and contains all the required information (such as the I2C bus and device address) for communicating with the sensor. The `device_is_ready` function verifies if the I2C bus is ready for use. If it is not

ready, the program will display an error message and exit.

- **Configuring the Sensor**

  The `buff1[]` array contains the registered address (`BH1749_MODE_CONTROL1`) and the data (`BH1749_MODE_CONTROL1_DEFAULTS`) to be written to it. `0x2A` configures the sensor with specific gains and measurement modes. The `i2c_write_dt` function sends the data in buff1 to the sensor over I2C. If the write fails, an error message is printed.

- **Enabling Measurement**

  In the code, the array `buff2[]` activates RGB measurements by adjusting bit 4 of the `BH1749_MODE_CONTROL2` register. The `i2c_write_dt` function transfers the data stored in `buff2` to the sensor. If the process is unsuccessful, an error message is recorded.

- **Main Loop: Reading and Printing Sensor Data**

  The array `rgb_value[6]` holds the RGB data obtained from the sensor. Each colour component (Red, Green, Blue) is represented by two bytes (LSB and MSB). The function `i2c_burst_read_dt` retrieves six bytes from the sensor, starting from the `BH1749_RED_DATA_LSB` register, and stores the RGB data in `rgb_value`. Bitwise operations are then used to calculate the red, green, and blue values by combining the LSB and MSB of each colour component. The RGB values are displayed on the console using `printk` statements. Finally, the program waits for `SLEEP_TIME_MS` (1 second) before starting the loop again, allowing for regular sensor readings.

This text outlines the setup and data reading procedures for the BH1749 RGB sensor on a Zephyr-powered microcontroller. It initializes the sensor, periodically retrieves the RGB values, and displays them on the console. I2C, device tree specifications, and Zephyr's kernel functions demonstrate the typical embedded development pattern on Zephyr RTOS.

The CMakeLists.txt file provided is a configuration file utilized by CMake, a build system generator, for building a project in Zephyr RTOS. Below is a detailed breakdown of each line in the file:

CMakeLists.txt

fund_less6_exer2 > CMakeLists.txt

```
#
# Copyright (c) 2016 Intel Corporation
#
# SPDX-License-Identifier: Apache-2.0
#

cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(Connecting_BH1749_Sensor)

target_sources(app PRIVATE src/main.c)

```

Figure 7.66: CMakeLists.txt for Connecting to the BH1749 Ambient Light Sensor on Thingy:53

- `cmake_minimum_required(VERSION 3.20.0)`

  The line sets the minimum `CMake` version the project needs to build. Specifically, it requires CMake version 3.20.0 or higher. If the system has an older version, the build process will terminate with an error indicating the version mismatch.

- `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})`

  instructs `CMake` to find the Zephyr project package essential for the build. The `REQUIRED` keyword means that failure to locate Zephyr will halter the build process with an error. The `HINTS` keyword guides `CMake` on where to search for Zephyr, utilizing the `ZEPHYR_BASE` environment variable to locate the Zephyr SDK. Typically, this environment variable is configured during the installation and setup of the Zephyr development environment.

- `project(Connecting_BH1749_Sensor)`

  This command labels the project as `Connecting_BH1749_Sensor`. The project name is utilized in generated build files and aids in organizing the build output. Generally, the name reflects the purpose or function of the project. This implies that the project focuses on connecting and interfacing with a BH1749 sensor.

- `target_sources(app PRIVATE src/main.c)`

  Specifies the source files for the build target.

  - `target_sources(app ...` denotes the addition of source files to the `app target`, representing the application being constructed.
  - `PRIVATE` keyword defines the scope of the added source files, indicating that they are exclusively used for building the specific target (`app`) and are not exposed to or reused by other targets.
  - `src/main.c` denotes the path to the source file within the project. In this case, `main.c` is in the `src` directory. The application's main source file contains the program's entry point and core logic.

This `CMakeLists.txt` file is simple and basic. It ensures that the necessary `CMake` version and Zephyr SDK are available, specifies the project name as `Connecting_BH1749_Sensor`, and includes the `src/main.c` file as the primary source code for the project. Running `CMake` in this project's directory will generate the required build files based on this configuration, allowing you to compile and build your Zephyr-based application.

Multithreaded Applications

nRF Connect SDK utilizes the Zephyr RTOS, a real-time operating system designed for embedded development. Zephyr comprises various kernel services and additional functionalities, such as threads, which facilitate the development of multi-threaded applications. This session will delve into the fundamental concepts of threads and the services offered by the nRF Connect SDK/Zephyr. As part of the exercise, we will explore the creation of threads with varying priorities and gain insight into the scheduler's behaviour, including features like time slicing and workqueue.

In this training, we aim to thoroughly comprehend the fundamental distinction between bare-metal and Real-Time Operating System (RTOS) programming by examining the advantages and disadvantages of utilizing an RTOS. We will also delve into the Zephyr RTOS execution model and gain familiarity with its

Interrupt Service Routines (ISRs), threads, thread lifecycle, inter-task communication/synchronization mechanisms, and scheduler.

Furthermore, we aim to grasp the basics of kernel services related to threads, including user-defined threads, system threads, and workqueue threads. In addition, we will explore preemptive scheduling and time-slicing through practical exercises, enabling us to understand how to create threads, perform thread yielding and sleeping, and offload work to a workqueue.

Thread Synchronization

Thread synchronization is necessary in applications with multiple threads running concurrently. This exercise will cover the importance of thread synchronization and the utilization of semaphores and mutexes as mechanisms for thread synchronization. The exercise section will examine two typical thread synchronization issues, demonstrating solutions using semaphores and mutexes [3].

This module aims to understand the critical need for thread synchronization mechanisms in concurrent programming. We will explore how multiple threads accessing shared resources can lead to data inconsistency and how thread synchronization mechanisms can help mitigate such issues. Additionally, we will delve into the fundamental properties of semaphores and mutexes, gaining insight into how they enable us to control access to shared resources among multiple threads. Through practical hands-on exercises, we will learn how to effectively use semaphores and mutexes for thread synchronization. We will gain valuable experience in ensuring the integrity of shared data in concurrent systems.

Semaphores

A semaphore, in its most basic form, is just a simple variable modified to indicate a shared resource's status. Semaphores serve as a means of sharing resources, where there is a finite amount of a resource that needs to be accessed by multiple threads. They primarily function as a signalling mechanism for controlling access to a specific number of resource instances. The following figure illustrates the properties of semaphores [3].

Semaphores exhibit the following characteristics: An initial count (greater than 0) and a maximum limit are established during initialisation. "Give" will increase

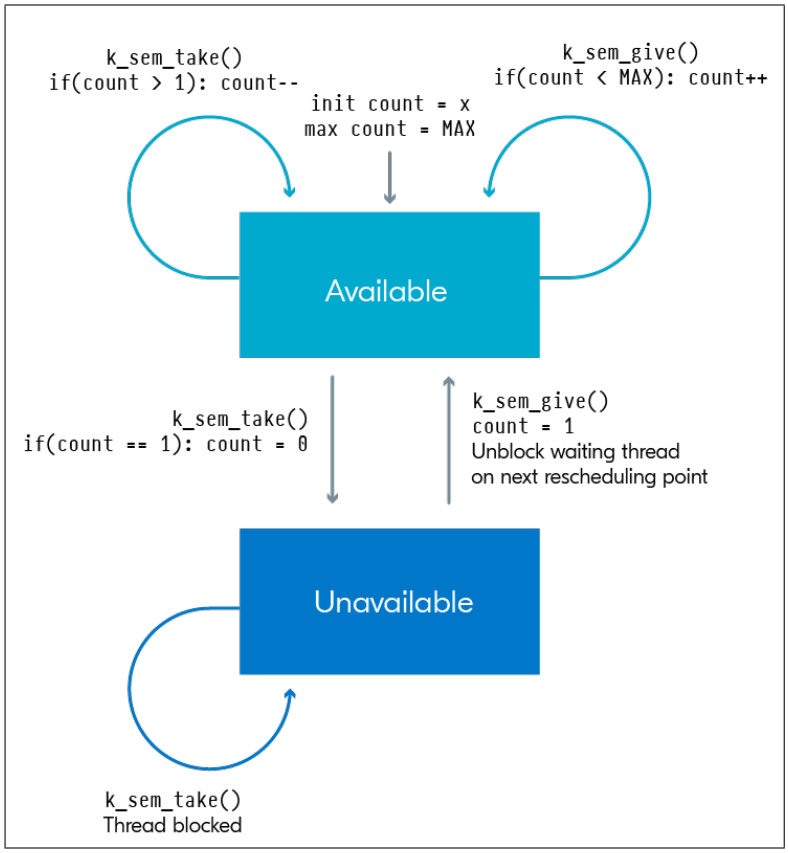


Figure 7.67: Semaphores [3]

the semaphore count unless the count is already at the maximum limit, in which case the signal will not increment. "Give" can be performed by any thread or ISR. "Take" will decrease the semaphore count unless the semaphore is unavailable (count at zero). Any thread attempting to take an unavailable semaphore must wait until another thread makes it available (by giving the semaphore). "Take" can only be executed by threads and not by ISRs (as ISRs should not be blocked by anything). There is no concept of ownership for semaphores. This implies that a semaphore can be taken by one thread and given by any other thread. It is not necessary for the thread that obtained the semaphore to be the one to release it. The thread that takes the semaphore is not entitled to priority inheritance, as the taking thread does not have ownership of the semaphore, and any other thread can release the semaphore [3].

The `prj.conf` file within a Zephyr project adjusts various build settings, en-

prj.conf 1 ×

fund_less8_exer1 > prj.conf

```
1   #
2   # Copyright (c) 2017 Linaro Limited
3   #
4   # SPDX-License-Identifier: Apache-2.0
5   #
6
7   CONFIG_PRINTK=y
8   CONFIG_ENTROPY_GENERATOR=y
9   CONFIG_TEST_RANDOM_GENERATOR=y
```

Figure 7.68: prj.conf for Semaphores

abling or disabling specific functionalities within the Zephyr kernel and its associated libraries. Each line in the `prj.conf` file corresponds to a configuration setting that alters the build process or the resulting firmware's behaviour. Below is an in-depth explanation of every configuration setting found in the provided `prj.conf` file:

- `CONFIG_PRINTK=y`

  Enabling the `printk` feature within the Zephyr project offers a lightweight, low-level logging method for debugging and diagnostic purposes. This feature permits the printing of messages to the console, commonly through a serial output, while the application is running. By configuring `CONFIG_PRINTK=y`, you guarantee the availability of `printk` in your code, allowing you to utilize it for displaying messages like debug information or status updates.

- `CONFIG_ENTROPY_GENERATOR=y`

  Enabling this option in Zephyr activates the entropy generator, which is essential for cryptographic operations, random number generation, and other security-related functions. The process involves creating high-quality random data. By setting `CONFIG_ENTROPY_GENERATOR=y`, your build will incorporate the necessary components to support entropy generation in your application. This is crucial if your application necessitates secure random numbers, such as cryptographic operations, unique identifiers, or nonces.

- `CONFIG_TEST_RANDOM_GENERATOR=y`

  The `CONFIG_TEST_RANDOM_GENERATOR` configuration enables a software-based random number generator for testing purposes. This feature is useful for scenarios where a hardware entropy source is unavailable, but random numbers are still required for testing or simulation purposes. It provides a basic random number generator that can be used in tests or non-production code where the quality of randomness is less critical.

The combination of these setups indicates that the project is designed to support fundamental debugging (using `printk`), creating entropy for security reasons, and a trial random number generator, perhaps for testing or non-essential applications.

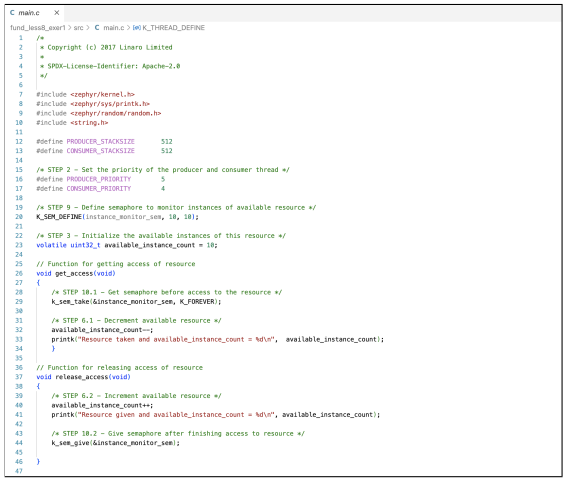


Figure 7.69: main.c part 1 - Semaphores

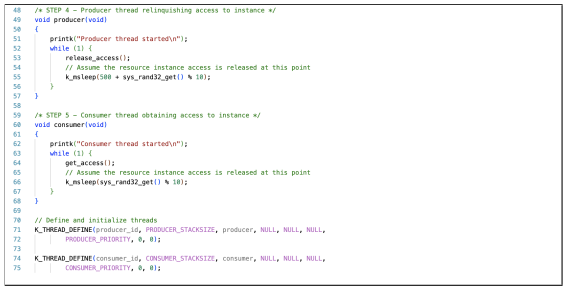


Figure 7.70: main.c part 2 - Semaphores

This `main.c` code implements a simple producer-consumer scenario using threads and semaphores in the Zephyr RTOS. It simulates resource management, where a producer releases a resource and a consumer acquires it. Below is a detailed explanation of each part of the code:

- **Include Statements**

  `<zephyr/kernel.h>` Contains the Zephyr kernel API and enables threading, synchronization, and timing functions. `<zephyr/sys/printk.h>` offers the `printk` function, a fundamental print function for debugging and logging messages. `<zephyr/random/random.h>` Encompasses APIs for producing random numbers are employed in this context to create variability in the timing of thread execution. `<string.h>` Provides standard `C` string functions, although not utilized in this code.

- **Thread Stack Sizes**

  The stack size for the producer and consumer threads is defined. Each thread is allocated 512 bytes of stack memory.

- **Thread Priorities**

  The priority of the producer thread is set to 5, while the priority of the consumer thread is set to 4. In Zephyr, lower numerical values correspond to higher priorities. This means that the consumer thread has a higher priority than the producer thread, resulting in its more urgent execution when both threads are ready to run.

- **Semaphore Definition**

  A semaphore is utilized to control access to a resource that has a limited number of instances. The semaphore, `instance_monitor_sem`, is initially set with a count of 10, signifying ten instances of the resource available. The maximum count is also restricted to 10, indicating that it cannot be increased beyond this value.

- **Resource Instance Count**

The variable `available_instance_count` is volatile, meaning that it keeps track of the number of available resource instances and can be modified by multiple threads. It is initialized to 10 to match the initial count of the semaphore. The volatile keyword prevents the compiler from optimizing our access to this variable.

- **Access Functions**

  In the `get_access()` function, the semaphore (`k_sem_take`) is acquired to access the resource. If no instances are available, the function will block the thread indefinitely (`K_FOREVER`). It also decreases the `available_instance_count` to indicate that an instance has been taken and logs the current count. The `release_access()` function increases the `available_instance_count` to indicate that an instance has been released and logs the updated count. It then releases the semaphore (`k_sem_give`), making an instance available again.

- **Producer Thread**

  The producer thread releases an instance of the resource by calling `release_access()` in an infinite loop. After each release, it sleeps for 500 ms plus a random time (between 0 and 9 ms) to simulate variability in the production rate.

- **Consumer Thread**

  The consumer thread continuously calls the `get_access()` function to obtain a resource instance. Upon obtaining the instance, it pauses for a random period, ranging from 0 to 9 milliseconds, to replicate fluctuations in consumption rates.

- **Thread Definitions**

  The macros in question create and set up the producer and consumer threads, represented by the identifiers `producer_id` and `consumer_id`. These threads are initialized with predefined stack sizes and priorities and set to begin execution immediately (with ‘0‘ as the final argument).

The following code demonstrates a scenario where two threads manage a limited resource with ten instances. One thread acts as a producer, releasing resource instances, while the other acts as a consumer, acquiring them. To regulate this process, semaphores ensure that the consumer cannot acquire more instances than are available, and the producer cannot release more instances than the maximum limit. Additionally, the system logs each action, providing visibility into the current resource availability for debugging and monitoring purposes.

Mutexes

Mutexes have only two states: locked or unlocked, unlike semaphores. They also have ownership properties, meaning only the thread that locks the mutex can unlock it. Think of a mutex as a mechanism that uses a single key for locking and unlocking. A thread must first acquire an unlocked mutex, lock it, and then access the object, such as a code section or a resource. If a thread trying to access the mutex finds it already locked, it will be blocked, so wait until the locking thread unlocks the mutex. A common use of a mutex is to protect a critical section of code that can be accessed by multiple threads, ensuring that the critical section is completed without interruption. The following graphic illustrates the properties of the mutex [3].

Mutexes have the following attributes: Locking a mutex will increment the lock count. Recursive locking will not cause the locking thread to block, as it already owns the mutex. The unlocking thread should ensure that it unlocks the mutex the same number of times it locks. Unlocking a mutex will decrease the lock count, and a count of zero indicates that the mutex is in an unlocked state, allowing other threads to attempt to acquire it. Only the thread that locked the mutex has the authority to unlock it. Mutex locking and unlocking operations are only performed by threads, not in Interrupt Service Routines (ISRs), as ISRs cannot participate in the scheduler's ownership and priority inheritance mechanism. The thread that locks the mutex is eligible for priority inheritance, as only that thread can unlock the mutex [3].

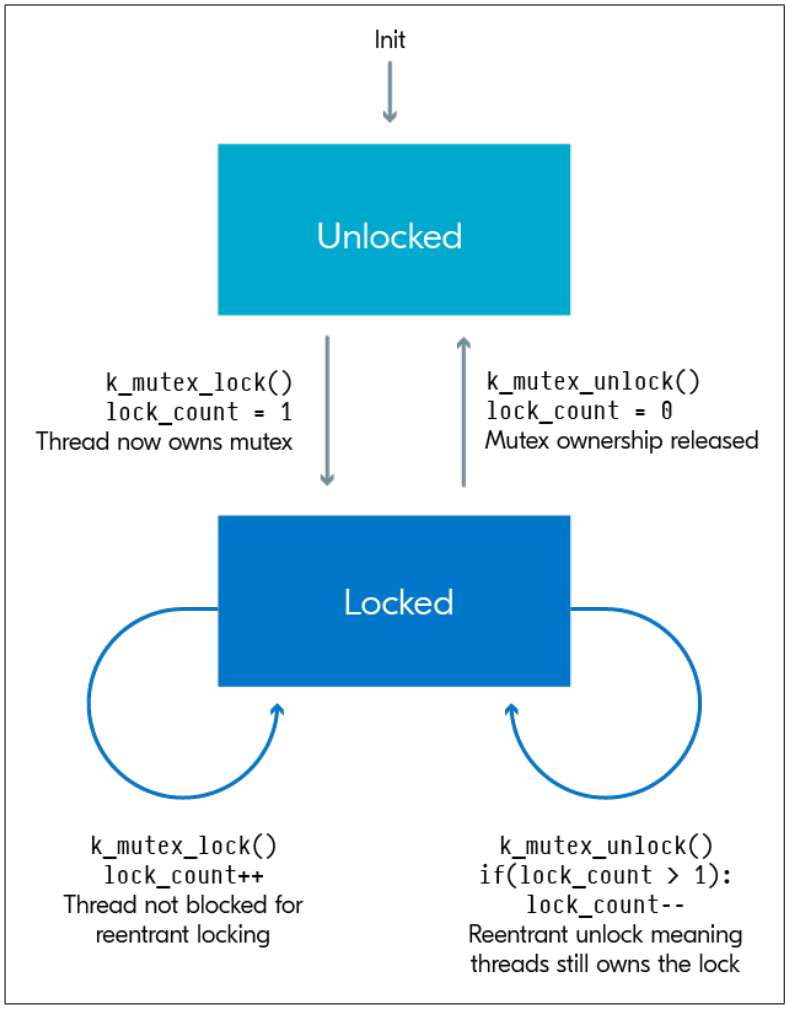
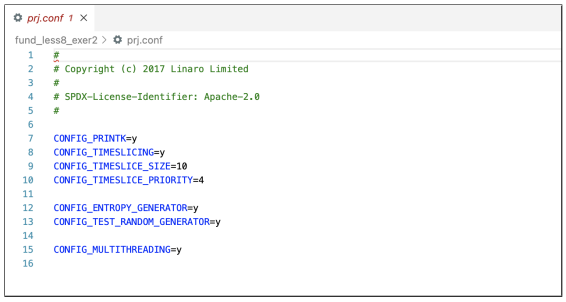


Figure 7.71: Mutexe [3]

nRF Connect for VS Code Extension Pack

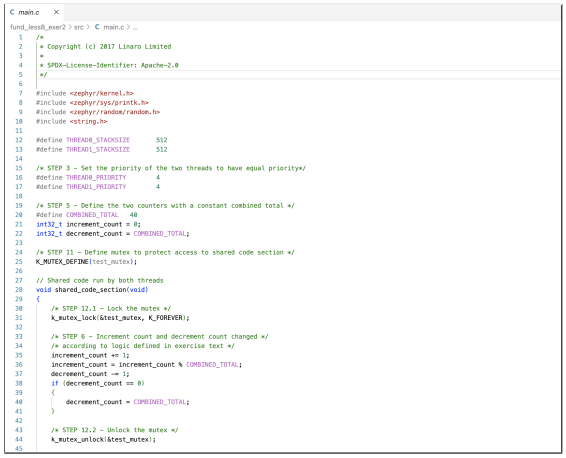
The extension pack nRF Connect for VS Code enables developers to utilize the widely used Visual Studio Code Integrated Development Environment (VS Code IDE) for creating, compiling, debugging, and deploying embedded applications based on Nordic's nRF Connect SDK. It provides access to the compiler, linker, entire build system, an RTOS-aware debugger, a seamless interface to the nRF Connect SDK, the Devicetree Visual Editor, and an integrated serial terminal, among other valuable development tools.

The nRF Connect for VS Code extension pack includes a range of components



```
prj.conf 1
fund_less8_exer2 > prj.conf
1   #
2   # Copyright (c) 2017 Linaro Limited
3   #
4   # SPDX-License-Identifier: Apache-2.0
5   #
6
7   CONFIG_PRINTK=y
8   CONFIG_TIMESLICING=y
9   CONFIG_TIMESLICE_SIZE=10
10  CONFIG_TIMESLICE_PRIORITY=4
11
12  CONFIG_ENTROPY_GENERATOR=y
13  CONFIG_TEST_RANDOM_GENERATOR=y
14
15  CONFIG_MULTITHREADING=y
16
```

Figure 7.72: prj.conf - Mutexes



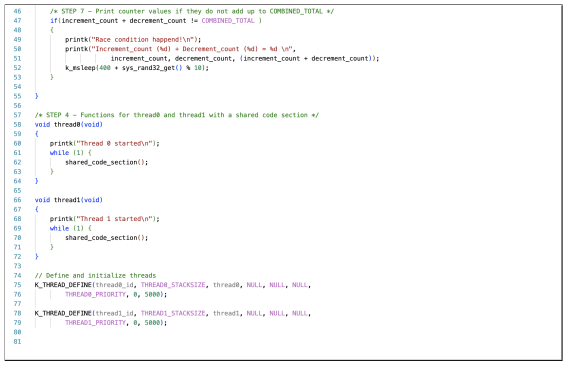
```
main.c
fund_less8_exer2 > src > main.c > ...
1   /*
2    * Copyright (c) 2017 Linaro Limited
3    *
4    * SPDX-License-Identifier: Apache-2.0
5    */
6
7   #include <zephyr/kernel.h>
8   #include <zephyr/sys/printk.h>
9   #include <zephyr/random/random.h>
10  #include <string.h>
11
12  #define THREAD0_STACKSIZE       512
13  #define THREAD1_STACKSIZE       512
14
15  /* STEP 3 - Set the priority of the two threads to have equal priority*/
16  #define THREAD0_PRIORITY        4
17  #define THREAD1_PRIORITY        4
18
19  /* STEP 5 - Define the two counters with a constant combined total */
20  #define COMBINED_TOTAL   40
21  int32_t increment_count = 0;
22  int32_t decrement_count = COMBINED_TOTAL;
23
24  /* STEP 11 - Define mutex to protect access to shared code section */
25  K_MUTEX_DEFINE(test_mutex);
26
27  // Shared code run by both threads
28  void shared_code_section(void)
29  {
30      /* STEP 12.1 - Lock the mutex */
31      k_mutex_lock(&test_mutex, K_FOREVER);
32
33      /* STEP 6 - Increment count and decrement count changed */
34      /* according to logic defined in exercise text */
35      increment_count += 1;
36      increment_count = increment_count % COMBINED_TOTAL;
37      decrement_count -= 1;
38      if (decrement_count == 0)
39      {
40          decrement_count = COMBINED_TOTAL;
41      }
42
43      /* STEP 12.2 - Unlock the mutex */
44      k_mutex_unlock(&test_mutex);
45
```

Figure 7.73: main.c part 1 - Mutexes

designed to enhance the functionality of the nRF Connect for VS Code platform. Firstly, the core extension, nRF Connect for VS Code, provides access to the build system and nRF Connect SDK. This extension facilitates the management of nRF Connect SDK versions and toolchains and allows for setting an active SDK/toolchain version.

The nRF DeviceTree component also supports the Devicetree language and the Devicetree Visual Editor, which benefits developers with complex hardware configurations. The nRF Kconfig component supports the Kconfig language, further enhancing the development environment's configuration.

Moreover, the nRF Terminal is a robust serial and RTT terminal, offering developers a streamlined way to interact with their devices. Regarding language support, the extension pack includes C/C++ from Microsoft, which integrates IntelliSense and other beneficial features for C/C++ development. Furthermore,



```
    /* STEP 7 - Print counter values if they do not add up to COMBINED_TOTAL */
    if(increment_count + decrement_count != COMBINED_TOTAL )
    {
        printk("Race condition happend!\n");
        printk("Increment_count (%d) + Decrement_count (%d) = %d \n",
                increment_count, decrement_count, (increment_count + decrement_count));
        k_msleep(400 + sys_rand32_get() % 10);
    }

}

/* STEP 4 - Functions for thread0 and thread1 with a shared code section */
void thread0(void)
{
    printk("Thread 0 started\n");
    while (1) {
        shared_code_section();
    }
}

void thread1(void)
{
    printk("Thread 1 started\n");
    while (1) {
        shared_code_section();
    }
}

// Define and initialize threads
K_THREAD_DEFINE(thread0_id, THREAD0_STACKSIZE, thread0, NULL, NULL, NULL,
        THREAD0_PRIORITY, 0, 5000);

K_THREAD_DEFINE(thread1_id, THREAD1_STACKSIZE, thread1, NULL, NULL, NULL,
        THREAD1_PRIORITY, 0, 5000);
```

Figure 7.74: main.c part 2 - Mutexes

including CMake and GNU Linker Map Files support further extends the capabilities of the development environment by offering language support and provision for linker map files.

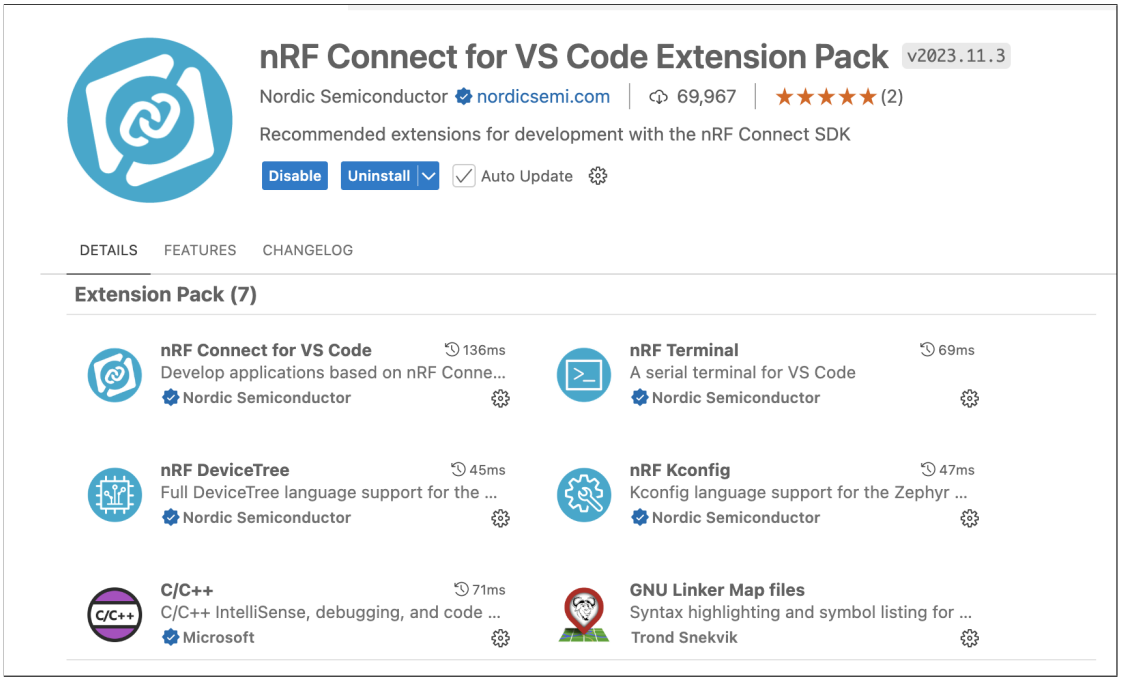


Figure 7.75: nRF Connect Visual Studio Code Extension Pack

Toolchain

The toolchain consists of multiple tools for constructing applications in the nRF Connect SDK. These encompass the assembler, compiler, linker, and CMake. Upon launching nRF Connect for VS Code for the first time, We will be asked to install a toolchain. This typically occurs if the extension cannot detect any installed toolchain on your device.

nRF Connect for Desktop Apps

- **Bluetooth Low Energy**

  This application makes it simple to perform Bluetooth low-energy connectivity testing across different platforms. It includes features for automatically detecting connected development kits, uploading firmware, and supporting Bluetooth Low Energy security functionality.

- **Board Configurator**

  The Board Configurator can modify the setup of Nordic Development kits (DKs), including virtual COM ports, supply voltage, SWD, and other settings.

- **Cellular Monitor**

  Cellular Monitor revolutionizes cellular development by offering real-time insights into modem behaviour, improving network assessment, and providing device information.

- **Direct Test Mode**

  The Bluetooth Core Specification shows that tests can be conducted using Direct Test Mode with Bluetooth low-energy devices. The application enables the configuration of devices as transmitters or receivers and adjusts settings such as channel, transmit power, and packet length.

- **nPM PowerUP**

  The PowerUP app by nPM links your PC to an nPM1300 EK, enabling code-free evaluation and development with the nPM1300 PMIC.

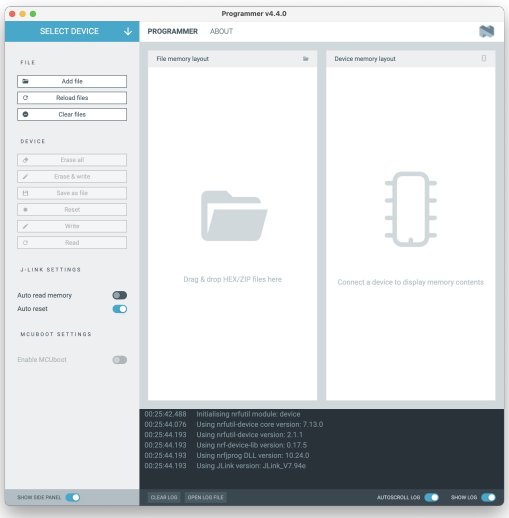


Figure 7.76: Programmer

- **Power Profiler**

  The Power Profiler App works with the Power Profiler Kit, allowing users to view, analyze, and export measurements. With the PPK2, we can perform dynamic current measurements ranging from 200nA to 1 A. Additionally, it can supply power to a connected board and measure its current.

- **Programmer**

  The Programmer app has been developed to simplify Nordic Semiconductor's System on Chips (SoCs) programming. Users can quickly transfer files by dragging and dropping them into the app's interface. Furthermore, users can perform actions like reading, writing, or erasing data on the connected device. The app is focused on making the programming of Nordic SoCs easier, providing convenience and efficiency to its users. nRF Connect Programmer is an application offered by nRF Connect for Desktop, enabling users to program firmware to Nordic devices. This application provides visibility into the memory layout for both J-Link and Nordic USB devices, allowing users to view the content of HEX files and write them to the devices.

- **Quick Start**

Quick Start provides a guided pathway for a faster and smoother evaluation and development experience.

- **RSSI Viewer**

  We can use the RSSI Viewer to explore the 2.4 GHz spectrum.

- **Serial Terminal**

  nRF Connect Serial Terminal is a terminal emulator used on different platforms to communicate with Nordic Semiconductor devices through a Universal Asynchronous Receiver/Transmitter (UART).

  The Serial Terminal lets you set up, monitor, and engage in virtual serial port communications with Nordic Semiconductor devices. It proves especially helpful for programming or debugging tasks, allowing you to observe logging outputs and input console commands. The terminal window supports shell mode for sending commands to devices running a shell-like Zephyr™ shell and line mode. It also supports ANSI escape codes, making it simple to locate warnings and errors.

  As an alternative to tools like PuTTY and minicom, Serial Terminal is tailored explicitly for Nordic Semiconductor devices. It features device autodetection, auto-reconnect, and persistence of device settings. Notably, terminal input and output are retained even after device disconnection. Serial Terminal facilitates shared access to a connected device's serial port with other compatible nRF Connect for Desktop applications.

- **Toolchain Manager**

  Maintain nRF Connect SDK and toolchain versions across Windows, Mac, and Linux.

Install Docker CE (Community Edition)

Following the correct installation instructions ensures everything runs smoothly when setting up your operating system.

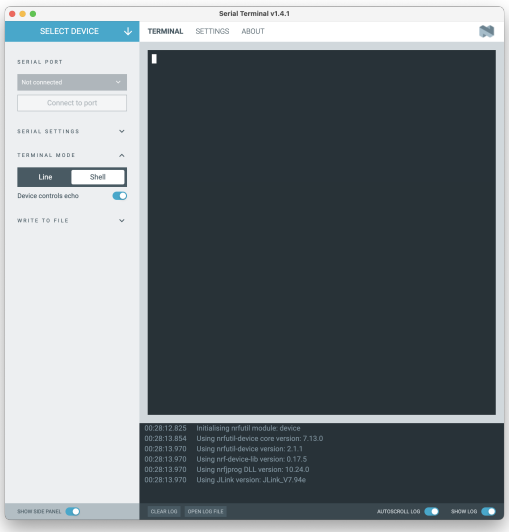


Figure 7.77: Serial Terminal

- For Linux users, we recommend following the installation guide for Docker for Linux, which can be found at:

  `https://docs.docker.com/desktop/install/linux-install/`

- If you are using MacOS, you can find our recommended installation instructions for Docker for MacOS at:

  `https://docs.docker.com/desktop/install/mac-install/`

- Lastly, if you are a Windows user, we advise following the Docker for Windows installation guide:

  `https://docs.docker.com/desktop/install/windows-install/`

Following the respective guidelines for your operating system will help you set up Docker appropriately.