LE Audio Test Infrastructure

Bachelor Thesis - Spring Term 2024

Department of Computer Science
OST - Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Authors
Jeremy Stucki & Vina Zahnd

Supervisor
Thomas Corbat, OST

Expert / Co-Examiner
Guido Zgraggen, Google, Inc. / Philipp Kramer, OST

Project Partner
Sonova Holding AG, Stäfa

13.06.2024

OST
Eastern Switzerland
University of Applied Sciences

sonova
HEAR THE WORLD

Abstract

The introduction of Bluetooth® Core Specification 5.2 has paved the way for new audio transmission features, like LE Audio and Auracast™, revolutionizing the audio technology landscape. This bachelor thesis, conducted in collaboration with Sonova Holding AG, aims to develop a test infrastructure to facilitate the testing of Auracast™ features in Bluetooth® LE Audio devices.

Given the limited adoption and support for Auracast™, there were not many existing solutions for testing available at the start of this thesis. The main goal of this thesis is to extend the existing test infrastructure of Sonova with a controlled environment for testing broadcast receivers, ensuring they can effectively receive and process Auracast™ broadcasts.

The test setup comprises two Nordic nRF5340 Audio Development Kits, configured as broadcast transmitter and assistant. A .NET-based library manages communication between the boards and the existing test infrastructure, allowing to set up an Auracast™ environment. Existing sample applications for the Nordic boards were translated from C to C++ and customized to meet the project's requirements. Additionally, a small console application was developed to test the setup locally.

The successful implementation of this project will contribute to the advancement of audio technologies, enhancing the use of end-user devices and enabling the development of innovative audio products. This thesis serves as a first step in adding Auracast™ capabilities to the test environment, which Sonova will continue to adjust to meet their evolving needs.

The future of Auracast™ appears promising, with many companies quickly adopting the trend. Some earbuds and smartphones already support the new features, and we are hopeful to see more widespread adoption in the near future.

Management Summary

The primary objective of this bachelor thesis is to extend Sonova's existing test infrastructure to facilitate the testing of Auracast™ features in broadcast receivers. Auracast™ is a wireless audio technology, introduced by Bluetooth® 5.2, that allows users to broadcast audio to multiple devices simultaneously, enhancing shared listening experiences in public or private settings. The setup includes a transmitter that sends the broadcast, receivers that listen to the broadcast, and an assistant that provides information about the available broadcasts. An example on how an Auracast™ setup could look like in a public area is shown in Figure 1.

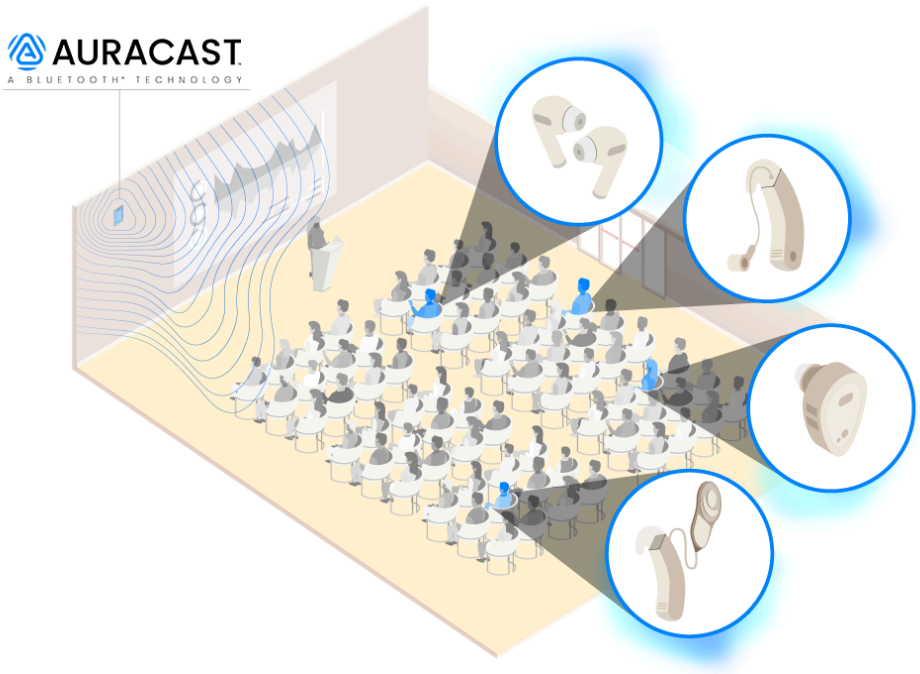

Figure 1: Image visualizing audio accessibility for all [1]

To stay competitive and integrate the latest technologies, Sonova wants to extend its current testing environment to support the new Auracast™ functionalities. The lack of existing infrastructure and applications for Auracast™ presents a challenge for companies looking to adopt this technology.

The project aims to develop a library that provides components to setup an Auracast™ environment and test receiver devices like ear buds or hearing aids. Key functionalities include:

- Providing at least one Auracast™ broadcast stream.
- Scanning for advertisement data to detect audio sink devices.
- Pairing and bonding with audio sink devices.
- Configuring the audio sink as a receiver of a broadcast stream through the broadcast assistant GATT service (BASS).

The project utilizes Nordic nRF5340 Audio Development Kits provided by Sonova for the broadcast transmitter and assistant roles.

Procedure and Technologies

The design phase involved evaluating several communication strategies, ultimately deciding on a simple, yet scalable solution using serial communication via UART. This approach balances immediate implementation needs with future expandability. Key design considerations included protocol selection (Protocol Buffers, text-based protocols, and JSON), with Protocol Buffers chosen for their efficiency and adaptability.

Before the actual implementation, experiments were carried out to determine what the possible options were. These were then presented to the Sonova team in a meeting, where it was discussed which one should be taken. The decision fell on an option that does not depend on already existing solutions at Sonova to have an independent solution. This included customizing the software loaded on the Nordic boards as well as controlling them.

After the main decision was made, the development process included setting up the repository, configuring the development environment, and implementing quality measures such as code reviews and continuous integration.

The project encountered various challenges, especially ones related to the embedded environment, as that was a new experience for the authors. Solutions involved pair programming, refactoring code, and collaborating with Sonova's embedded software team to resolve issues.

Results and Outlook

In this thesis, a basic test setup was successfully created with which broadcast sinks can be tested, an overview of the solution is visualized in Figure 2.

The colored lines of the diagram should be interpreted as follows:

- Black Lines: Library Commands
- Gray Lines: Sonova Test System Communication
- Dotted Lines: Wireless Communication

The setup includes all key functionalities mentioned above but is missing some of the more advanced features and configurability. After the completion of this bachelor thesis, there will be a handover process. The integration into the test system will be carried out by another department and is therefore not part of this thesis. The project will be continued by Sonova and adapted according to requirements.

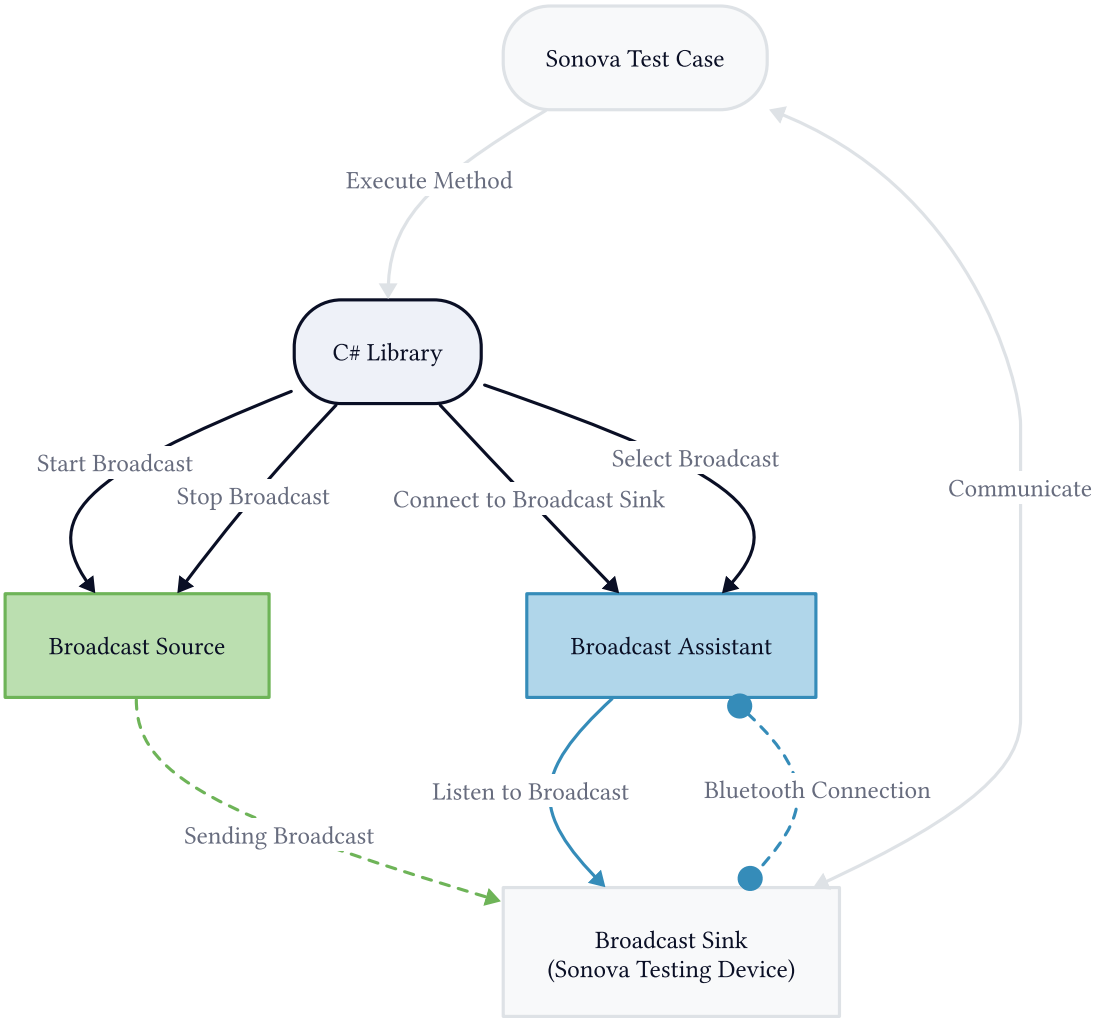

Figure 2: Diagram visualizing the implemented solution

Table of Contents

1. Introduction

For this bachelor thesis, we collaborated with the external partner Sonova AG.

> Sonova is a global leader in innovative hearing care solutions: from personal audio devices and wireless communication systems to audiological care services, hearing aids and cochlear implants. The Group was founded in 1947 and is headquartered in Stäfa, Switzerland.
>
> — Sonova [2]

The Bluetooth® Core Specification 5.2 introduced new features like LE Audio and Auracast™ to support different audio transmission scenarios. They are advertised as

> "The next generation of Bluetooth® audio - Building on 20 years of innovation, LE Audio enhances the performance of Bluetooth® audio, adds support for hearing aids, and introduces Auracast™ broadcast audio, an innovative new Bluetooth® use case with the potential to once again change the way we experience audio and connect with the world around us."
>
> — Bluetooth® [3]

A typical Auracast™ setup includes essential components like the audio sink for receiving and rendering audio signals, the broadcast transmitter emitting audio signals for consumption, and optionally, the broadcast assistant aiding in broadcast discovery and selection.

In the development and testing of audio devices interfacing with Auracast™ broadcasts, the availability of controllable counterparts within the test environment is crucial. This thesis aims to establish the infrastructure needed to configure test scenarios for audio devices receiving Bluetooth® Auracast™ broadcasts, aiming to contribute to the evolution of audio technology and enhance user experiences in the digital realm.

1.1. Initial Situation

The introduction of Auracast™, a new Bluetooth® specification, has seen limited adoption by companies so far. As of the start of this project, only a few smartphones on the market already supported Auracast™, including the Samsung S23 and S24 Series [4]. As a result, there is a significant lack of applications and libraries that support it, presenting both opportunities and challenges for our project.

Currently, many companies are starting to look into using this new part of Bluetooth®, which opens up the opportunity for exploration and innovation, paving the way for the development of new products. The anticipated benefits of the new features promise significant improvements, allowing users to broadcast audio to multiple devices simultaneously and enhancing shared listening experiences in public or private settings.

As Auracast™ becomes more popular, it is exciting to think about how it will grow. During our bachelor thesis, we are interested in seeing how companies will start to support Auracast™ and LE Audio, and what this will mean for the future of this technology.

1.2. Problem Description

Sonova aims to keep pace with the latest technologies on the market. To incorporate Auracast™ into their new products, they require a method for testing its functionality. An existing test environment from Sonova, containing automated test cases, should be extended to support testing new Auracast™ features of their hearing aids. For testing the added features, an Auracast™ environment needs to be set up, which automated tests can manage through the newly developed library.

1.3. Project Goal

This section describes the goals of this project according to the task assignment *Section 18.3, Assignment*.

This bachelor thesis aims to extend Sonova's existing test infrastructure to facilitate the testing of Auracast™ features in broadcast receivers. Specifically, the project will focus on supporting test scenarios involving one or, ideally, multiple Auracast™ broadcasts, a broadcast assistant, and broadcast receivers.

Within the test environment, control over the broadcast transmitter(s) and the broadcast assistant is needed. Controlling the receivers (system under test) is outside the scope of this project, as they can already be controlled by the existing test infrastructure. The specific feature set was elaborated during the project and was adapted according to the project's progress. This was done in collaboration with Sonova and especially their embedded software team.

The developed components must include the following essential features: providing at least one Auracast™ broadcast stream, scanning for advertisement data to detect audio sink devices, pairing/bonding with audio sink devices, and configuring the audio sink as a receiver of a broadcast stream through the Broadcast Audio Scan Service (BASS).

The device under test could be any Bluetooth® LE Audio/Auracast™ sink, like off-the-shelf headphones supporting it. For the broadcast transmitter and assistant Sonova suggests nRF5340 Audio Development Kits (DKs) [5] and provides the corresponding hardware.

While the current environment for testing devices requires the implementation of a .NET-based library for controlling the external devices, integration into other environments should be possible as well. This should be considered when designing the system.

1.4. Structure of This Report

This report encompasses the analysis, design, and implementation of the project's work. It is structured into the following sections:

Section 2, Analysis: Analysis which we performed of Bluetooth® technologies, especially Audio-related ones. We also took a look at the DK and its capabilities.

Section 3, Requirements: Functional and non-functional requirements that we derived from the assignment given.

Section 4, Initial Experimentation: Details the experiments we conducted with the DKs during the early stages of this project.

Section 5, Design: Contains various decisions we had to make, such as communication strategy and serial protocol. Here we also define the interaction of the components.

Section 6, Development Process: Details our repository setup and development environment. Also contains a listing of the libraries we used and discusses the quality measures we took.

Section 7, Implementation: Details the implementation of the library and the two embedded applications. Also contains the detailed protocol definitions.

Section 8, Verification: Reviews the previously defined requirements and describes our approach to automated and manual testing.

Section 9, Challenges: Describes challenges we encountered during the development phase.

Section 10, Project Management: Explains the approach we took for this project, our project plan, and various time tracking reports.

Section 11, Conclusion: Describes the conclusion of our project. This section also contains the learnings we made, as well as an outlook on the future.

2. Analysis

Within this section we describe the Bluetooth® technologies used for Auracast™. In *Section 2.1, Bluetooth® LE (BLE)* we document what it is and of what it contains. *Section 2.2, LE Audio* describes the LE Audio specification and explains the new Auracast™ feature. The DK we used is introduced in *Section 2.3, Nordic nRF5340 Audio DK*. Finally *Section 2.2.1.1, Supported Assistant and Receiver Devices* explains which devices can currently be used to test Auracast™.

2.1. Bluetooth® LE (BLE)

Bluetooth® Low Energy (BLE) was introduced in 2010 as part of the Bluetooth® 4.0 Specification. Before BLE was introduced, there was only Bluetooth® Classic, both protocols are covered by the Bluetooth® Specification and operate on the 2.4 GHz ISM band. The two protocols are two distinct and incompatible protocols. Most modern smartphones are dual-mode Bluetooth® devices, supporting both BLE and Bluetooth® Classic at the same time. [6]

The difference between the two is that Bluetooth® Classic is great in handling a lot of data, but also uses a lot of energy doing that. A Traditional use cases is transmitting files like music or images.

BLE on the other hand is designed for applications that require good battery life and do not need to transfer a lot of data. This would be a good fit for things like sensors. [7].

Figure 3 displays the described wireless technologies and visualizes their differences.

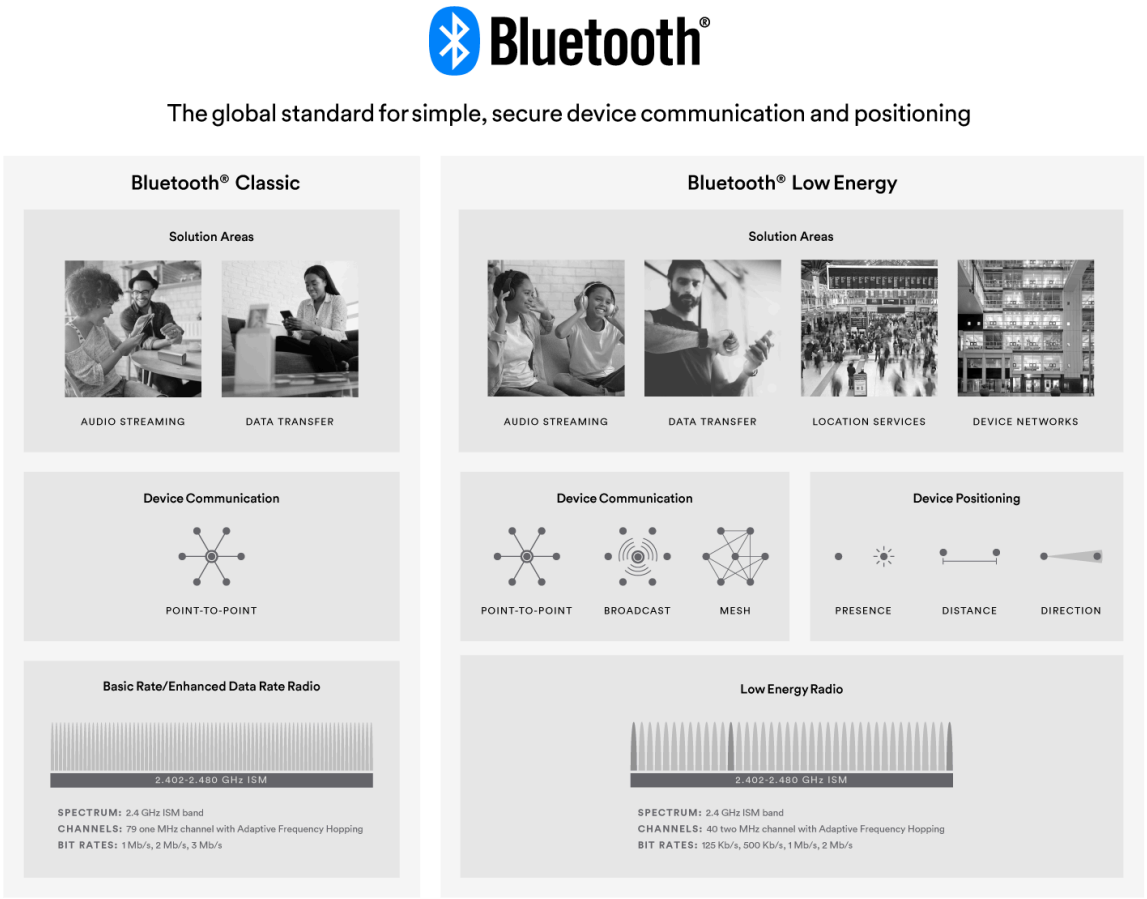

Figure 3: Comparison of Bluetooth® Classic and BLE [8]

2.1.1. Generic Access Profile (GAP)

Generic Access Profile (GAP) provides a framework that defines how the BLE devices interact with each other, controlling connections and advertisements. This includes the roles of BLE devices, advertisements and connection establishment. GAP defines four main roles that a BLE device operates in. [9] Table 1 shows the different roles and what they do.

Central	Device that **scans** and **initiates** connections with peripherals. **Examples:** Smartphone, Smart Home Hub [10]
Peripheral	Device that **advertises** its existence and **accepts** connections from centrals. **Examples:** IoT Device, Heart Rate Monitor, Temperature Sensor, Fitness Tracker [10]
Broadcaster	Special peripheral device that **sends** advertisements without the capability to accept connection requests from a central. **Examples:** Beacon devices
Observer	Special central device that **discovers** peripherals and broadcasters without the capability to accept connection requests from a central.

Table 1: Roles defined by the Bluetooth® GAP

The key distinction between two BLE devices in "connected" mode versus "advertising-discovery" mode is that the "connected" mode allows bi-directional data transfer between the devices. In contrast, a device in advertising mode (peripheral or broadcaster) cannot receive any data from an observer or central device. [9]

2.1.2. Advertisements

In the advertising state the device sends out packages to make itself visible to other devices. These packages contain useful data for the receivers to process.

There are 40 Radio Frequency (RF) channels in total, separated by 2MHz. Figure 4 displays the advertising channels in BLE. Channels 37, 38 and 39 are the Primary Advertising Channels, the other remaining channels are called the Secondary Advertisement Channels. [9]

Secondary Advertisement Channel Before a device can advertise itself on a secondary advertising channel, it first needs to advertise on a primary advertising channel. [9]

Primary Advertising Channel If a device wants to utilize the secondary advertising channel, it needs to send advertising packets on the primary channel that then point to secondary advertising packets. [9]

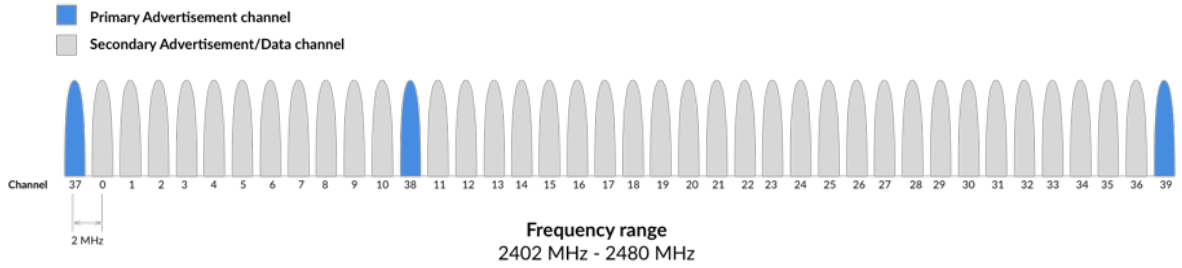

Figure 4: Advertising channels in BLE [9]

2.2. LE Audio

Before the introduction of LE Audio, there was only Classic Audio, which operates on the Bluetooth® Classic radio. Figure 5 illustrates the two types of Bluetooth® audio: Classic Audio and LE Audio.

Figure 5: Types of Bluetooth® audio [11]

LE Audio brings new features, while still supporting the audio products and use cases of Classic Audio, such as wireless headphones, speakers, and in-car entertainment systems. These new features promise to enhance performance and enable the development of new products and applications. [12]

Classic Bluetooth® technology primarily supports Synchronous Connection-Oriented (SCO) Streams, which are isochronous and used for telephony and voice assistant functions through the Hands-Free Profile (HFP). For unidirectional audio transmission, asynchronous Audio Streams are used.

The introduction of broadcasts is a feature that was first implemented in BLE. Additionally, the idea of a headphone pair consisting of two independent devices is a development that came about with the advent of LE Audio. Figure 6 shows ear buds connected to a smartphone using Connected Isochronous Streams (CIS) and Connected Isochronous Groups (CIG).

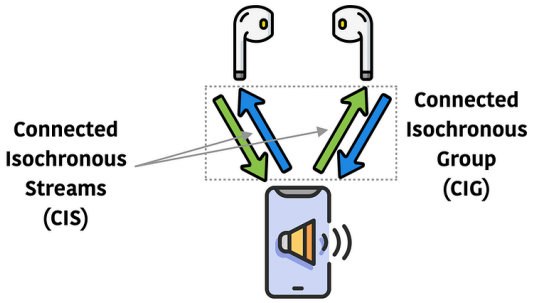

Figure 6: Connected Isochronous Streams (CIS) and Connected Isochronous Groups (CIG) [13]

The future of BLE devices looks very promising when looking at the market research from ABI [11].

> "While Bluetooth® Classic has dominated the market historically, in recent years, many audio device manufacturers have begun to adopt dual-mode Bluetooth® Classic and LE radio solutions.
>
> While Bluetooth® Classic is typically used to stream audio, Bluetooth® LE can be used for faster pairing, media control, and to enable location functionality in order to track earbuds. Most leading Bluetooth® wireless chipset vendors addressing the audio market today offer dual-mode radio solutions (i.e., Bluetooth® Classic + Bluetooth® LE) as part of their product portfolios. Over time, these dual-mode radio solutions will increasingly support LE Audio functionality and become dual-mode audio solutions (i.e., Classic Audio + LE Audio). This will help to enable new Auracast™ broadcast audio use cases while allowing vendors to continue to innovate on their product offering via additional features and performance differentiators."
>
> — Andrew Zignani [11]

The research also mentions that dual-mode audio devices will dominate the market, as shown in the shipment forecasts through 2027 in Figure 7.

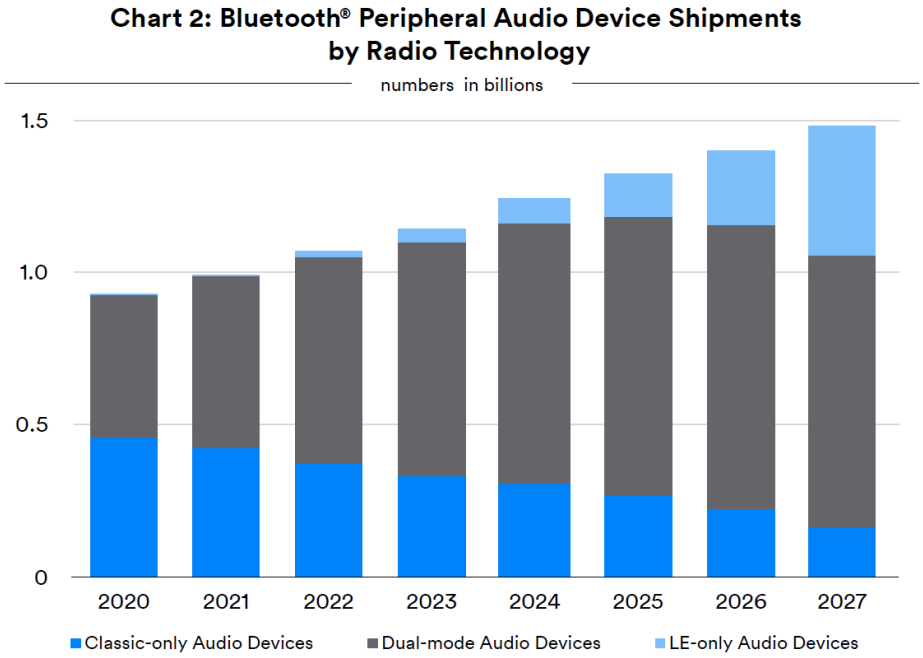

Figure 7: Projected Bluetooth® audio device shipments by supported audio modes [11]

2.2.1. Auracast™ (Broadcast Audio)

Auracast™ was introduced as part of the Bluetooth® 5.2 Specification in 2020. Auracast™ is not a Bluetooth® standard, but a set of requirements utilizing the Public Broadcast Profile (PBP) specification within Bluetooth® LE Audio. PBP is part of the Bluetooth® LE Audio specification, allowing for a universal format and the availability of public broadcasts.

The idea behind Auracast™ is that a device can start one or multiple broadcast streams, to which every device that supports Auracast™ as a receiver could listen to, without undertaking a pairing process.

One use case for this could be public speech where the speaker is using a microphone. The input of the microphone can then be streamed and people using hearing aids, or headphones can choose to listen to this stream so they can hear the speaker loud and clearly.

The standard consists of three main parts:

- Transmitter
- Assistant
- Receiver

The transmitter starts an Auracast™ broadcast and sends advertisements, which provide information for the assistant. The broadcast can have multiple channels, such as stereo left/right, different quality levels, or different languages. The Auracast™ assistant then is able to see the advertisements and can present information about the available streams to the user. Next, the user can select which steam they want to listen to and finally, the assistant sends the instruction to the receiver to listen to the given stream. [14]

The names transmitter, assistant and receiver are not Bluetooth® roles on their own, instead they were introduced to describe their part in the Auracast™ ecosystem. They are made up of many underlying roles, encompassing roles from the Basic Audio Profile and other higher-level specifications. In Table 2 these underlying roles are shown.

Figure 8 visualizes the components of Auracast™ and gives an idea on how it works.

Name	Includes the role of	Specification
Auracast™ transmitter	• Broadcast Source • Public Broadcast Source • Initiator • Broadcast Media Sender	• BAP • PBP • CAP • TMAP
Auracast™ receiver	• Broadcast Sink • Public Broadcast Sink • Acceptor • Hearing Aid • Broadcast Media Receiver	• BAP • PBP • CAP • HAP • TMAP
Auracast™ assistant	• Broadcast Assistant • Public Broadcast Assistant • Commander	• BAP • PBP • CAP

Table 2: Underlying specification roles covered by the Auracast™ terminology [15]

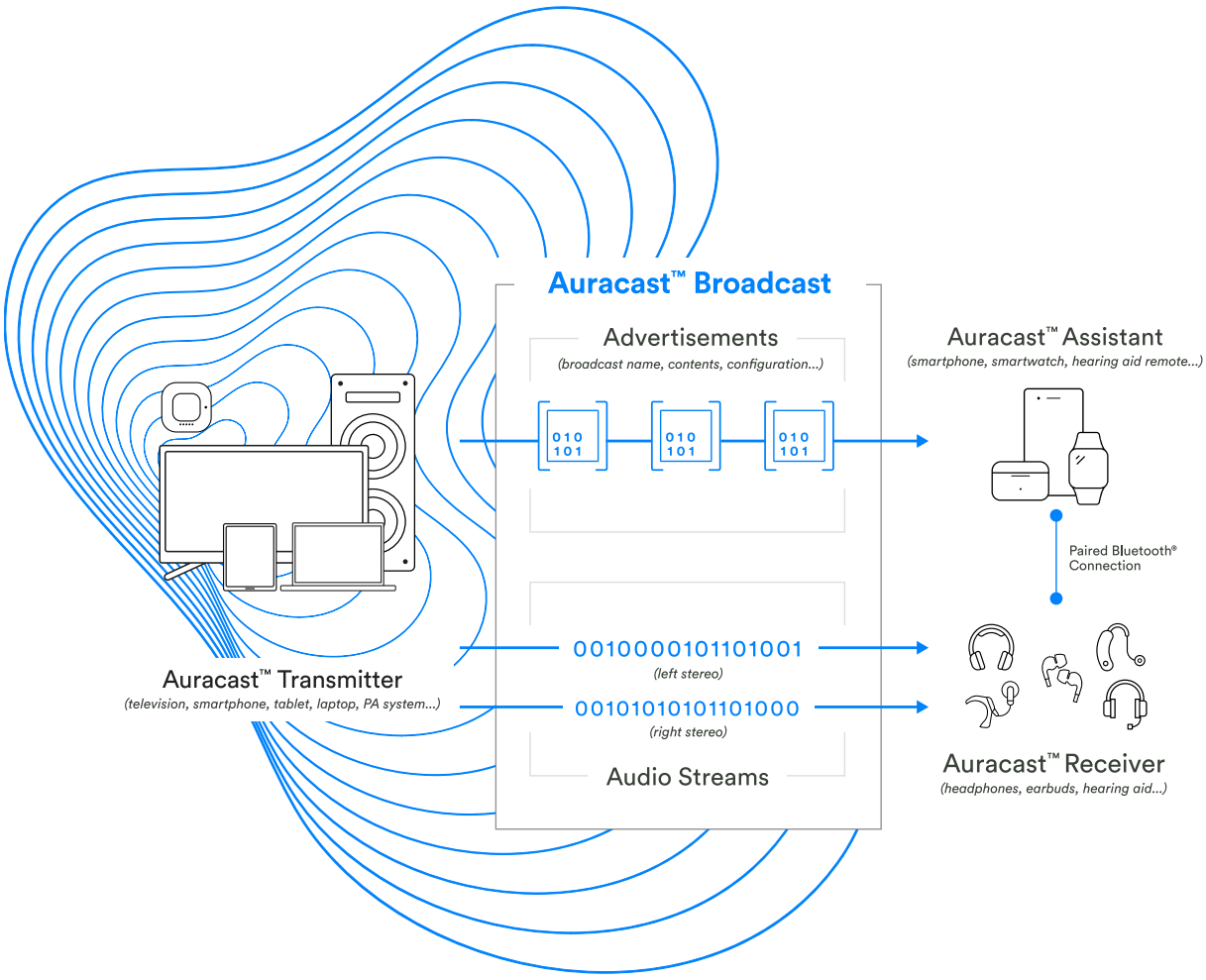

Figure 8: How Auracast™ works [14]

2.2.1.1. Supported Assistant and Receiver Devices

For a device to act as an assistant or receiver, it needs to implement a set of specifications defined in the Bluetooth® LE Audio specification, as shown in Table 2.

An assistant could be an application on a phone, a remote control, a smartwatch, or some other product.

A receiver needs an assistant to know what broadcasts are available and to receive the information needed to listen to one of them. It is possible for a receiver to also include an assistant, but as earbuds or hearing aids are usually equipped with a limited amount of user interaction features, such as buttons, they can use an external device instead.

When a pair of receivers is used, such as earbuds, hearing aids, or speakers, each piece listens to a stream. The assistant chooses which stream they will receive, this could be the same or for example, one stream for the left and one for the right side. Receivers are not sending acknowledgments for packages received from the transmitter, as this would lead to unnecessary traffic. [16]

Many currently available headphones already support Bluetooth® 5.2 or higher but lack the LE Audio support. As of March 15, 2024, we could not find any official applications for Android or iOS smartphones that support Auracast™. However, according to a press release from Samsung [4] from February 20, 2024, they added support for listening to broadcasts in version 5.1.1 of One UI, and support for starting audio broadcasts in version 6.1 of One UI. Version 6.1 of One UI, which supports both features, is available on Galaxy S24 series, S23 series, Z Fold5, Z Flip5, and Tab S9.

Table 3 lists all the headphones we discovered, that already support Auracast™. Even though the product listing for the Galaxy Buds 2 Pro does not explicitly list Auracast™ in the specifications [17], they are able to listen to broadcast streams. We were able to test the Galaxy Buds2 Pro as a receiver and can therefore conclude that this device supports Auracast™.

Product Name	**Brand**	**Bluetooth® Features**
Galaxy Buds 2 Pro [17]	Samsung	Bluetooth® 5.3 & LE Audio
Momentum True Wireless 4 [18]	Sennheiser	Bluetooth® 5.4, LE Audio and Auracast™

Table 3: Headphones supporting Bluetooth® 5.2 or higher and LE Audio

2.3. Nordic nRF5340 Audio DK

For testing we received Nordic nRF5340 Audio DKs, which support BLE and Auracast™ features. The board has configurable buttons and LEDs. It can be connected to the computer through USB. [5] In this project the idea is to make these boards accessible to the library, which is used in the test environment.

Applications for the boards can be built with the nRF Connect Software Development Kit (SDK) [19], which is based on Zephyr [20]. Detailed information about the device can be found on the website of Noridc Semiconductor [5].

The key features of the board that are needed, that we can use them for this project include:

- Bluetooth® 5.2
- Support of all Auracast™ features
- Configurable
- Powered via USB

We need to have at least two devices, one acting as a broadcast and one as an assistant. A third one acting as a receiver is not needed, as this part would be the device under test from Sonova. The board itself is very handy and easy to transport, an image of the Nordic nRF5340 Audio DK is shown in Figure 9.

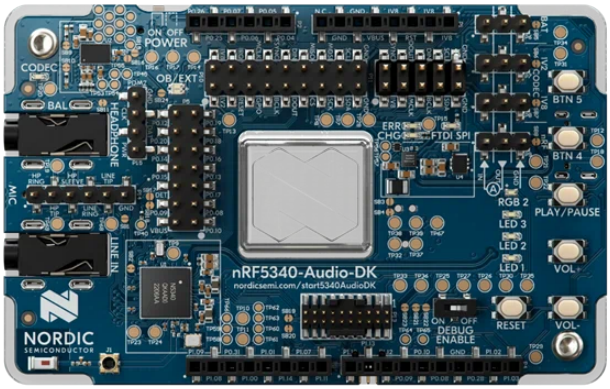

Figure 9: Nordic nRF5340 Audio DK [5]

2.4. Bluetooth® Generic Attribute Profile (GATT)

Generic Attribute Profile (GATT) is an acronym for Generic ATTribute Profile [21], defining a service framework using the Attribute Protocol (ATT). [21] This framework defines procedures and formats of services and their characteristics, including procedures for discovering, reading, writing, notifying, indicating, and broadcasting them. [22]

GATT is used after establishing a dedicated connection between two devices, following the advertising process managed by GAP (*Section 2.1.1, Generic Access Profile (GAP)*). Connections are exclusive, meaning that a BLE peripheral device can only be connected to one central device at a time. The peripheral device will stop advertising itself as soon as it is connected to a central. [21] Other devices will not be able to see or connect to the peripheral device.

2.4.1. Attribute Protocol (ATT) - Roles

The protocol defines two roles, server and client.

Server The server is the device storing data that can be read or written by the client. This could be a fitness tracker that stores the latest heart rate value on the server.

Client The client is the device requesting data from the server. This could be a smartphone connected to a fitness tracker.

These two ATT roles are independent of the roles defined by GAP, meaning a device that supports a GAP central role will not automatically assume the client ATT role. [23]

2.4.2. Attribute Protocol (ATT) - Attributes

The server device functions as a database to store data that it needs to share with the client device. The exposed data is structured as attributes, which contain an attribute handle, attribute type (Universally Unique Identifier (UUID)), attribute value and attribute permissions field. [23] Figure 10 shows the structure of an attribute.

Attribute Handle This is a unique unsigned 16-bit identifier used by the client to reference an attribute on the server, making it "addressable". The does not change during a single connection.

Attribute Type (UUID) This globally unique 2-byte or 16-byte UUID identifies the type and meaning of data in the attribute's value field, with two types: Service UUID and Characteristic UUID.

Attribute Value This is the actual data being stored.

Attribute Permission Field This specifies the various methods that can be used to access the attribute value, as well as the security level required to access it.

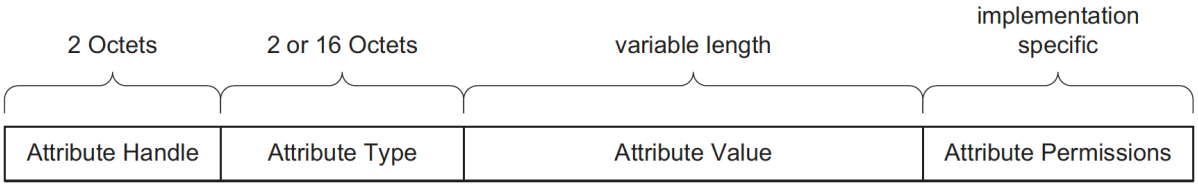

Figure 10: Attribute structure [24]

Besides the classic method to access the data, using write and read operations initiated by the client (Figure 11), it is also possible for the server to notify the client when data has changed (Figure 12). In this case the server would need to send a notification operation, sending an updated value of an attribute to the client every time it changes. The server also sends an indication operation similar to the notification operation, but the client needs to respond with an acknowledgment status, indicating if the value has been successfully received. [23]

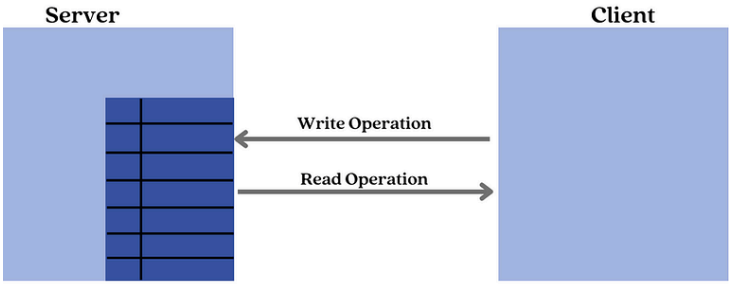

Figure 11: Write and read ATT-defined access methods [23]

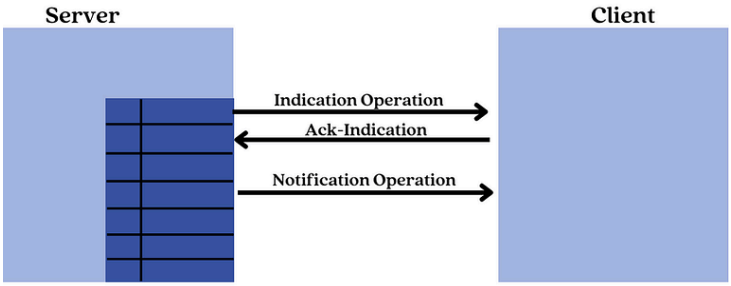

Figure 12: Notification and indication ATT-defined access methods [23]

3. Requirements

Functional requirements are described in *Section 3.1, Functional Requirements*. Non-functional requirements can be found in *Section 3.2, Non-Functional Requirements*.

Requirements essential for the Minimum Viable Product (MVP) are indicated with the **MVP** label.

3.1. Functional Requirements

The functional requirements are split into the three subgroups “Broadcast Source”, “Broadcast Assistant” and “Library”. To make it easy to differentiate them by their ID, we use a convention:

FR1xx Broadcast Source Requirements
FR2xx Broadcast Assistant Requirements
FR3xx Library Requirements

3.1.1. Broadcast Source

ID	**FR101** MVP
Title	Starting and Stopping the Stream
Description	A broadcast stream can be started and stopped.

Table 4: FR101 – Starting and Stopping the Stream

ID	**FR102**
Title	Multiple Streams
Description	Multiple streams can be configured/started/stopped.

Table 5: FR102 – Multiple Streams

ID	**FR103**
Title	Naming the Streams
Description	A name can be chosen for each stream.

Table 6: FR103 – Naming the Streams

ID	**FR104** MVP
Title	Playing Test Tone
Description	A test tone can be played, so that no audio source needs to be provided.

Table 7: FR104 – Playing Test Tone

ID	**FR105**
Title	Audio Input Selection
Description	Audio input can be selected from multiple sources and is sent over the stream.

Table 8: FR105 – Audio Input Selection

ID	**FR106**
Title	Encrypted Streams
Description	Broadcast transmissions can be encrypted with a key.

Table 9: FR106 – Encrypted Streams

3.1.2. Broadcast Assistant

ID	FR201 MVP
Title	Scanning for Sink Devices
Description	The assistant can scan for available sink devices and can query them for GATT services to determine which ones support Auracast™.

Table 10: FR201 – Scanning for Sink Devices

ID	FR202 MVP
Title	Pairing of Specific Device
Description	A sink can be paired by providing the necessary connection information.

Table 11: FR202 – Pairing of Specific Device

ID	FR203 MVP
Title	Connecting to Specific Device
Description	A sink can be connected after the pairing process has finished.

Table 12: FR203 – Connecting to Specific Device

ID	FR204 MVP
Title	Selecting Stream and Sending to Sink
Description	A specific stream can be selected and its configuration is sent to the sink.

Table 13: FR204 – Selecting Stream and Sending to Sink

ID	FR205
Title	Scanning for Broadcast Sources
Description	The assistant can scan for broadcast sources and list them.

Table 14: FR205 – Scanning for Broadcast Sources

ID	FR206
Title	Scanning for Broadcast Sinks
Description	The assistant can scan for broadcast sinks and list them.

Table 15: FR206 – Scanning for Broadcast Sinks

3.1.3. Library

ID	**FR301**
Title	Receiving GATT Information
Description	The library needs to provide information from GATT.

Table 16: FR301 – Receiving GATT Information

3.2. Non-Functional Requirements

The non-functional requirements are split into subgroups according to Functionality, Usability, Reliability, Performance and Supportability (FURPS). To make it easy to differentiate them by their ID, we use a convention:

NFR1xx NFRs related to Functionality
NFR2xx NFRs related to Usability
NFR3xx NFRs related to Reliability
NFR4xx NFRs related to Performance
NFR5xx NFRs related to Supportability

3.2.1. Functionality

ID	**NFR101** **MVP**
Title	Comply with Industry Standards
Description	Coding conventions and guidelines should be used if they are provided.
Acceptance Criteria	Sonova coding guidelines are used.
Method of Verification	Code review

Table 17: NFR101 – Comply with Industry Standards

ID	**NFR102** **MVP**
Title	Dependency Management
Description	Current versions of dependencies should be used.
Acceptance Criteria	When a dependency is added, it must be the latest version available.
Method of Verification	Code review

Table 18: NFR102 – Dependency Management

3.2.2. Usability

ID	**NFR201** **MVP**
Title	Graceful Error Handling
Description	The library reacts in a predictable way to errors and reports them correctly upstream.
Acceptance Criteria	In case of an error, the application using the library is never presented with a generic error message. The error messages forwarded inform the application about the source of the error.
Method of Verification	Unit tests & Code review

Table 19: NFR201 – Graceful Error Handling

ID	**NFR202** **MVP**
Title	Abstraction Level
Description	A good abstraction level should be used, so that the underlying communication layer can easily be replaced.
Acceptance Criteria	An interface exists, allowing the replacement of the communication layer without having to adjust the call-site.
Method of Verification	Code review

Table 20: NFR202 – Abstraction Level

ID	**NFR203** **MVP**
Title	Ease of Use
Description	A software engineer can use the library by just looking at its documentation.
Acceptance Criteria	• Annotations are used on public methods, describing their functionality and behavior. • Methods and variables are named appropriately. • A good readme is in place.
Method of Verification	Code review

Table 21: NFR203 – Ease of Use

ID	**NFR204** **MVP**
Title	Usage Examples
Description	There exists usage examples.
Acceptance Criteria	At least one usage example is present.
Method of Verification	Code review

Table 22: NFR204 – Usage Examples

3.2.3. Reliability

ID	**NFR301**
Title	Reliable Operation
Description	The library operates reliably under various conditions, so that no-one needs to touch the test setup after a power-cycle.
Acceptance Criteria	• Recovers from power interrupts • Can deal with hot-plugging of devices
Method of Verification	Manual test

Table 23: NFR301 – Reliable Operation

3.2.4. Performance

ID	**NFR401** MVP
Title	Startup Time
Description	A stream should be started relatively quickly.
Acceptance Criteria	Streams can be started within less than 300ms if the correct firmware is already present on the device.
Method of Verification	Manual test

Table 24: NFR401 – Startup Time

ID	**NFR402**
Title	Performance of Multiple Streams
Description	Starting multiple streams has a minimal effect on performance
Acceptance Criteria	No stream breaks down or has dropped frames.
Method of Verification	Manual test

Table 25: NFR402 – Performance of Multiple Streams

3.2.5. Supportability

ID	**NFR501** MVP
Title	Integration of Library
Description	The integration of the library should be easily possible.
Acceptance Criteria	• The library can easily be integrated into the existing test environment. • A brief and understandable readme instruction text is providing a step by step guidance to integrate the library.
Method of Verification	User test

Table 26: NFR501 – Integration of Library

ID	**NFR502** MVP
Title	Maintainability
Description	The codebase should be easy to maintain and extend.
Acceptance Criteria	A new software engineer can take over maintenance of the library without any verbal instructions, by just looking at the readme and source code.
Method of Verification	User test

Table 27: NFR502 – Maintainability

ID	**NFR503**
Title	Clear Versioning System
Description	A clear versioning system is used.
Acceptance Criteria	Version numbers follow the “MAJOR.MINOR.PATCH” format.
Method of Verification	Code review

Table 28: NFR503 – Clear Versioning System

ID	**NFR504**
Title	Platform Agnostic Implementation
Description	Ensure compatibility with various operating systems and platforms where tests may run.
Acceptance Criteria	An appropriate abstraction level is used, so that the adoption to a new testing infrastructure is made possible.
Method of Verification	Code review

Table 29: NFR504 – Platform Agnostic Implementation

4. Initial Experimentation

To familiarize ourselves with the Nordic DKs, we implemented a basic setup. In this configuration, we used one board as a transmitter, which received audio through the USB interface and transmitted it to the second board. The second board then received the audio data and played it through the headphone output. For this experimentation we used the provided C examples with only minor modifications. The setup used is depicted in Figure 13.

By following guides from Nordic Semiconductor, we successfully set up the Nordic DKs. Although the guides were detailed, their organization made them somewhat difficult to follow. For example, while consulting the instructions for the `buildprog.py` utility [25], we initially believed we had all the necessary command-line arguments. However, we encountered issues with programming the boards. We later discovered two additional arguments mentioned in a different section. Notably, the `-p` argument is essential for programming the boards.

We were able to view the serial output of the two DKs using the configuration found in the testing and optimization documentation [26].

During this experiment, we learned how to work with the boards and which tools are required to develop applications for them.

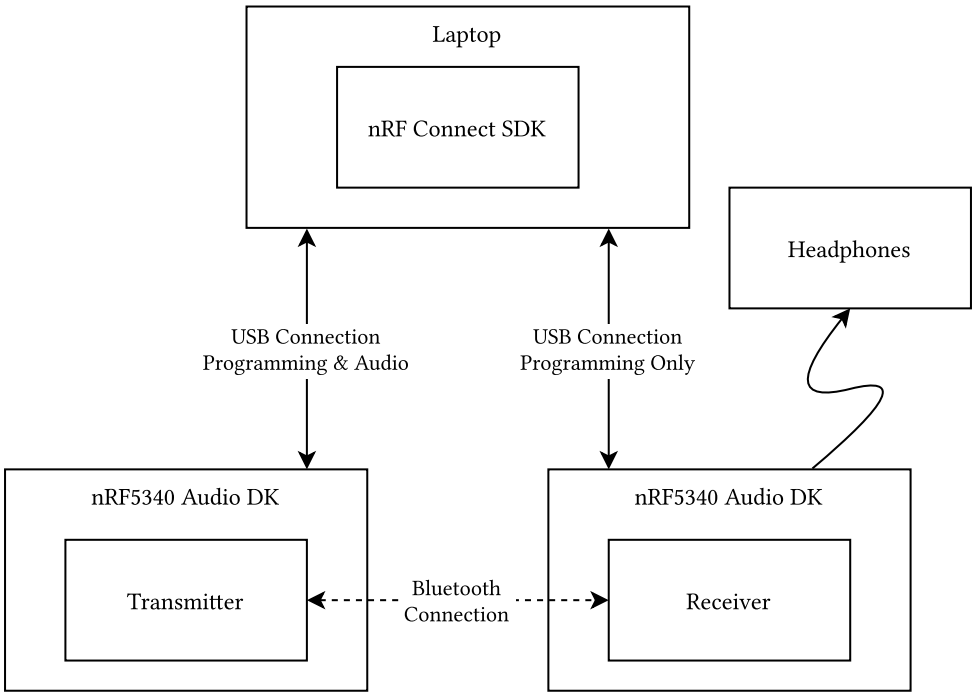

Figure 13: Setup used for initial experimentation

5. Design

This section outlines the various design strategies we considered and the decisions we made throughout the process. *Section 5.1, Overview* starts out by giving a high-level overview over the parts involved and their relationship. In *Section 5.2, Communication Strategy*, we explore the high-level architecture of the individual components and the technology used. In *Section 5.3, Serial Protocol*, we define the serial protocol for our chosen communication strategy, ensuring efficient and reliable communication between components. The messages sent over the defined serial protocol are defined in *Section 5.4, Component Interaction*. Finally, in *Section 5.5, Components*, we provide a more detailed look at the individual components.

5.1. Overview

As a result of our analysis described in *Section 2, Analysis*, we created a diagram shown in Figure 14 to show the individual parts of the project and how they interact with each other. This forms the basis for our design considerations, which we will explore in the following sections.

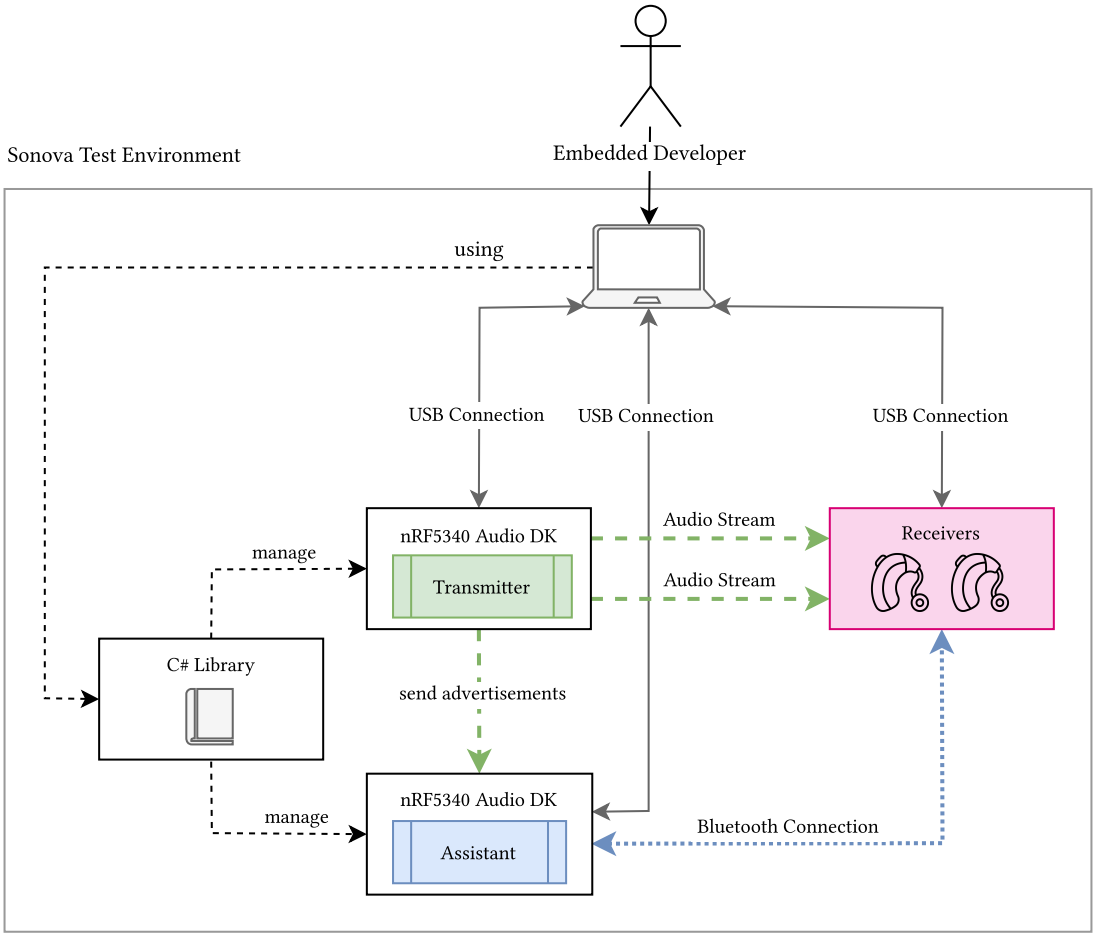

Figure 14: Working environment [27]

5.2. Communication Strategy

There are several methods for facilitating communication between our library and the development board, each with its own set of advantages and disadvantages. In this section we will examine three distinct approaches and then decide on one of those for further development.

Option A: In *Section 5.2.1, Option A - Serial Communication via UART*, we explore communication over a simple serial interface.

Option B: In *Section 5.2.2, Option B - HCI Bridge*, we consider exposing the entire Host Controller Interface (HCI) interface to the library

Option C: In *Section 5.2.3, Option C - Existing Sonova Implementations*, we analyze existing implementations that could be adapted.

5.2.1. Option A - Serial Communication via UART

In this variant, we utilize serial communication to establish a connection with the board. The advantage here lies in our ability to use existing sample applications that already handle all necessary hardware interactions. We would need to adopt the existing C samples to C++ and make them configurable. Additionally, we would need to write clients that handle the communication layer.

However, this strategy also includes notable drawbacks. We must define the communication channel ourselves, adding a layer of complexity to the project. Moreover, developing a custom Zephyr application in C or C++ becomes a necessity.

An architectural overview for this strategy is depicted in Figure 15. A basic proof of concept implementation can be found in *Section 5.2.1.1, Proof of Concept*.

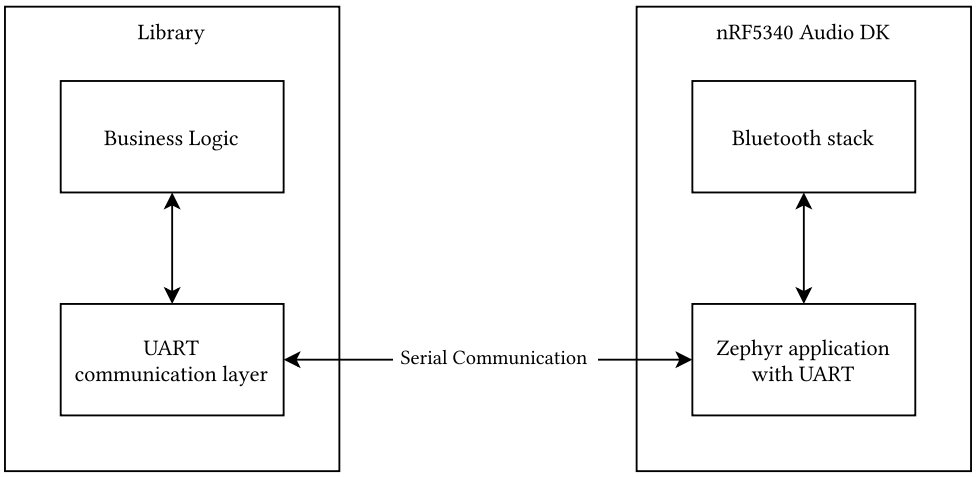

Figure 15: Communication strategy - Architectural overview of serial communication via Universal Asynchronous Receiver-Transmitter (UART)

5.2.1.1. Proof of Concept

In the proof of concept, we revisited the fundamentals by examining examples from the Zephyr project and intentionally overlooked the nRF Connect SDK, as it seemed to be quite a few versions behind Zephyr and was lacking the new Auracast™ examples. The documentation for the Zephyr project was exceptionally well-written and facilitated easy setup.

We modified an existing sample application, specifcally the “broadcast source” sample application to include UART interrupt support, which enables communication with the application on the board. This example application starts a broadcast stream that Auracast™ receivers can listen to. In this proof of concept, we only introduced a single command, which, when pressing the letter “s”, pauses or resumes the broadcast.

Additionally, we wrote a basic .NET script in C# that sends the command to the board through the Component Object Model (COM) interface. This is shown in Listing 1.

```
using System.IO.Ports;

var serialPort = new SerialPort();
serialPort.PortName = "COM4";
serialPort.BaudRate = 115200;

serialPort.Open();
serialPort.Write("s");
```

Listing 1: Communication strategy - Sample code of UART communicaton variant

5.2.2. Option B - HCI Bridge

In this variant we utilize the HCI protocol, which gives full access to the Bluetooth® controller to our library. This would bring the major advantage that the boards would both have identical software and the logic of broadcast and assistant would live in our library. It also means that the hardware can easily be exchanged for other hardware without much adjustment, as an industry-standard protocol is used.

Since writing the entire Bluetooth® logic ourselves is out of the scope of this project, this strategy relies on having a reliable library that abstracts the HCI communication for us. Figure 16 shows how this would look like.

Unfortunately, we were not able to find a suitable library, so this strategy would involve writing a lot of low-level code. The libraries we analyzed can be found in *Section 5.2.2.1, Analysis of HCI Libraries*.

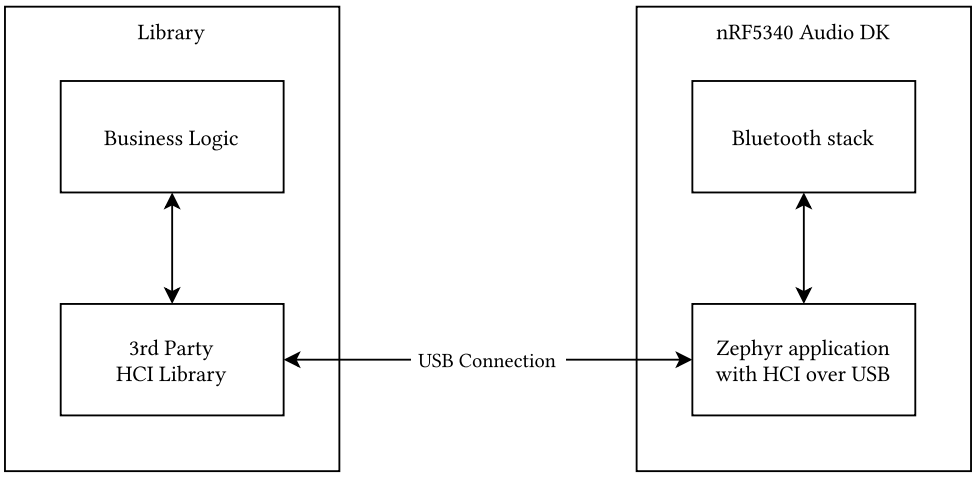

Figure 16: Communication strategy - Architectural overview of HCI Bridge

5.2.2.1. Analysis of HCI Libraries

We looked at various libraries, but unfortunately, none of them were suitable for our use case. Some smaller ones exist as well, but we would want a well-maintained one in order to have a good foundation. Table 30 shows the bigger libraries we considered.

Library	**Reasoning**
bumble by Google [28]	The library seemed like a good fit at first, however it is still in alpha and therefore too unstable for our use case.
Windows Bluetooth® Driver [29]	The driver enables a very low-level communication, however this would require us to write a big chunk of the abstraction we need ourselves and is therefore not the right tool for us.

Table 30: Bluetooth® HCI libraries we looked at

5.2.3. Option C - Existing Sonova Implementations

For this option, we looked at existing internal implementations, which we could adapt or reuse. This took considerable time to research, but we can only give a high-level overview due to a non-disclosure agreement.

We communicated with multiple teams within the company to get to know which possibilities we had. We also looked at the source code of these existing solutions to figure out what they could and could not do.

The existing solutions we analyzed were all tightly integrated into bigger projects and it would have been non-trivial to extract the part that we would use. The level of abstraction was also not really the one we were looking for.

5.2.4. Decision Process

This chapter delves into the decision-making process concerning communication strategy options within Sonova's dedicated team. The focus is on achieving a balance between simplicity and functionality, ensuring both immediate effectiveness and future adaptability.

In the meeting with the Sonova team, the discussion centered on the optimal approach to designing the communication strategy between the library and the Nordic boards. It was unanimously agreed that prioritizing simplicity and functionality over unnecessary complexity is crucial for ensuring the long-term utility of the library. The emphasis was on developing a solution that not only meets immediate needs but also remains adaptable for future expansions.

One big concern was getting things up and running quickly without sacrificing the ability to grow later on. They thought about incorporating some elements of option B described in *Section 5.2.2, Option B - HCI Bridge*, like using direct HCI commands instead of calling functions on the board. This would make it easier to expand the library in the future.

Additionally, the team stressed the importance of avoiding the need to reflash the operating system on the boards for simple configuration changes such as bitrate adjustments. To address this, configurations should be managed within a separate configuration file, utilizing formats like JavaScript Object Notation (JSON) or Protocol Buffers for simplicity and ease of maintenance.

Furthermore, there was consensus on the desirability of incorporating a User Interface (UI) element. This UI would facilitate manual testing and comprehension of receivers, offering functionalities, such as start/stop streams and real-time board status updates (e.g., "connected" or "connection lost"). This addition is seen as instrumental in enhancing usability and aiding in the troubleshooting processes.

In navigating the decision process, the Sonova team prioritizes a pragmatic approach that balances immediate needs with future-proofing considerations. By emphasizing simplicity, adaptability, and user-centric design, the chosen communication strategy aims to optimize efficiency and effectiveness in the development and maintenance of the library.

5.2.5. Decision

All participants of the meeting agreed, that the best solution is to go for option A which is described in *Section 5.2.1, Option A - Serial Communication via UART*. In this case a simple solution can be created within a short time. The solution should be built to be easily adjusted and can be developed further after the bachelor thesis is completed.

Some employees also hinted that using multiple stacks is beneficial, so re-using an existing one might not even have been that great. This is to test their end-devices on multiple stacks, which is a good thing for when these devices ultimately get used in the real world, where each manufacturer might implement certain things his own way.

To tighten this decision we created a weighed decision matrix with the key criteria mentioned in the meeting. Table 31 illustrates a weighted decision matrix according to the discussion.

		Option A Serial Communication via UART		**Option B** HCI Bridge		**Option C** Existing Sonova Implementations	
Criteria	**Weight**	**Rating**	**Total**	**Rating**	**Total**	**Rating**	**Total**
Functionality	10%	2	6.67%	3	10%	1	3.33%
Complexity	20%	3	20%	1	6.67%	2	13.33%
Configurability	5%	1	1.67%	3	5%	2	3.33%
Extendability	30%	2	20%	3	30%	1	10%
Development Time	20%	3	20%	1	6.67%	2	13.33%
Stability	10%	2	6.67%	3	10%	1	3.33%
Reuse of Existing Tools	5%	2	3.33%	1	1.67%	3	5%
Total	**100%**		**78.33%**		**70%**		**51.67%**

Table 31: Weighed decision matrix for communication strategies

5.3. Serial Protocol

To communicate with the board we have a simple UART connection over a serial interface. In order for us to send instructions to the applications running on the board, we need to define a protocol to use on top of it. We looked at three different options and compared them on a few key metrics. The conclusion is described in *Section 5.3.5, Decision*.

5.3.1. Option A - Protocol Buffers

The first option is to use Protocol Buffers, which allow specifying messages in a platform-agnostic way and are then using source generation to generate encoders and decoders. [30] It allows to transfer the data in a small but not human readable format. A big plus point of Protocol Buffers is that they are language agnostic, which enables the same data structure to be used in a different project if needed. Also, the Sonova team told us that they were already using Protocol Buffers within one of their projects and therefore some knowledge is already established.

5.3.2. Option B - Text-Based Protocol

The second option is writing a custom text-based protocol. This involves writing a specification and then implementing it in C# and C++, which gives us full control over the communication channel. Despite the flexibility it would take quite a good amount of time to implement a good text-based protocol. As this would be a custom protocol, it would also need to be adapted to other applications manually.

5.3.3. Option C - JSON

The third option is using JSON, for which a lot of encoders and decoders already exist, however the resulting structure still needs to be mapped to appropriate types on either end. [31] What makes JSON a good option is that it is used almost everywhere and is a common protocol to use and therefore the knowledge of how to use it is widely spread. The protocol is easy to understand and implement but it does not define the contained data structure, so a parser is still required to be written for each language, which increases the implementation time.

5.3.4. Decision Process

To decide on which serial protocol we should use, we evaluated key metrics and weighted them accordingly.

5.3.4.1. Communication Overhead

Using a text-based protocol can be more efficient than JSON, but still comes way short when compared to the binary Protocol Buffers. This mostly boils down to the fact that even JSON stripped of all unnecessary whitespace still has the overhead of field labels. Protocol Buffers mostly transmit the actual content of the fields with just minimal overhead for field identification.

Winner: Protocol Buffers
Runner up: Text-based protocol

5.3.4.2. Readability

Having a human readable protocol can aid in the development process, allowing us to use a serial monitor to view messages being transmitted. Both the text-based protocol, as well as JSON can be read at a glance without having a decoder.

Winners: Text-based protocol & JSON

5.3.4.3. Development Time

With Protocol Buffers we can skip a whole lot of decoding and parsing work compared to the other options. JSON still has an edge over the text-based protocol, since at least there the parsing does not have to be done from scratch.

Winner: Protocol Buffers
Runner up: JSON

5.3.4.4. Future Adaptability

With Protocol Buffers source code can be generated in various programming languages, removing the need to re-implement a decoder and parser. With JSON the parsing can be done by a supporting library, but mapping the resulting structure to proper types would still need to be done.

Winner: Protocol Buffers
Runner up: JSON

5.3.5. Decision

Based on the categories we looked at we took a decision on how to continue. We weighted all categories equally and came to the conclusion that Protocol Buffers are the way forward. The only downside of Protocol Buffers is that they are not human-readable, which we will compensate for by implementing a good logging infrastructure. Table 32 illustrates the process.

		Option A Protocol Buffers		**Option B** Text-Based Protocol		**Option C** JSON	
Criteria	**Weight**	**Rating**	**Total**	**Rating**	**Total**	**Rating**	**Total**
Communication Overhead	20%	3	20%	2	13.33%	1	6.67%
Readability	20%	1	6.67%	3	13.33%	2	20%
Development Time	20%	3	20%	1	6.67%	2	6.67%
Future Adaptability	40%	3	30%	1	10%	2	20%
Total	**100%**		**76.67%**		**43.33%**		**53.34%**

Table 32: Decision matrix for the serial communication protocol

5.4. Component Interaction

In this section we defined the interaction between the boards and the library at a high level. This served as a basis for the implementation described in *Section 7, Implementation*, where we also decided on the message content. We defined what commands are needed for the solution to work and which response is expected for each of them. We defined commands which are needed to establish an Auracast™ setup, except for the broadcast sink as this part will be the test subject of Sonova.

5.4.1. Setting the Color

A simple use case is setting the color of the LED on the nordic board. Having different colors for broadcast source and assistant can help to differentiate between the two, especially during development. When a reset is needed, the reset button of the correct device can be pressed. This command is not necessarily needed, but is helpful to have and easy to implement. Both implementations of the boards need to implement this feature so it can be set on both.

Figure 18 visualizes the steps needed to set the color on the broadcast source and assistant.

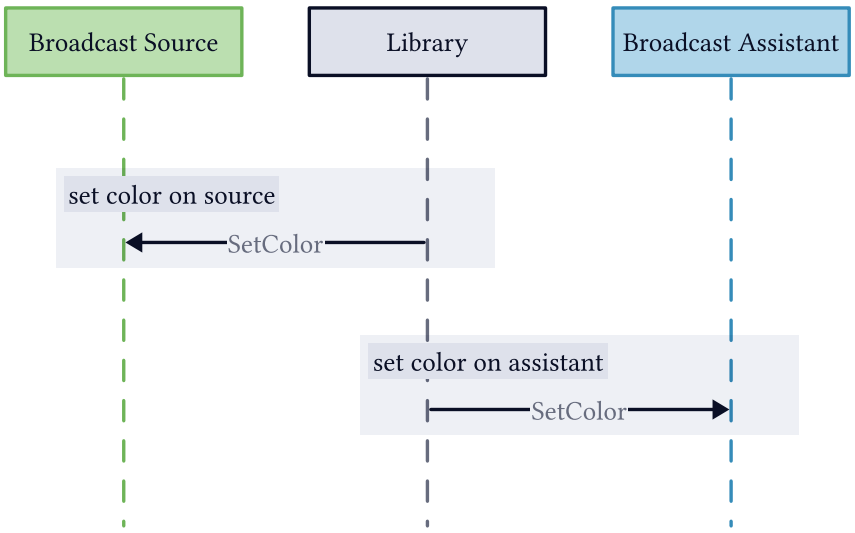

Figure 17: Library design - Setting the LED colors

5.4.2. Starting the Broadcast

For the broadcast source to be visible to the assistant it needs to advertise itself. It also needs to stream sound for the broadcast sink to listen to. Both can be handled within one command to keep the interaction with the broadcast source as simple as possible.

Figure 18 visualizes the steps needed to start broadcasting.

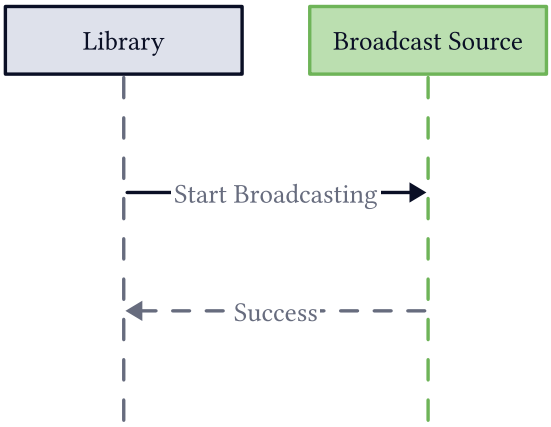

Figure 18: Library design - Starting the broadcast

5.4.3. Stopping the Broadcast

When a broadcast source is not needed to be discovered and/or listened to it can simply be stopped by sending the stop broadcasting command. The broadcast source will stop to advertise itself and stop the broadcasts.

Figure 19 visualizes the steps needed to stop broadcasting.

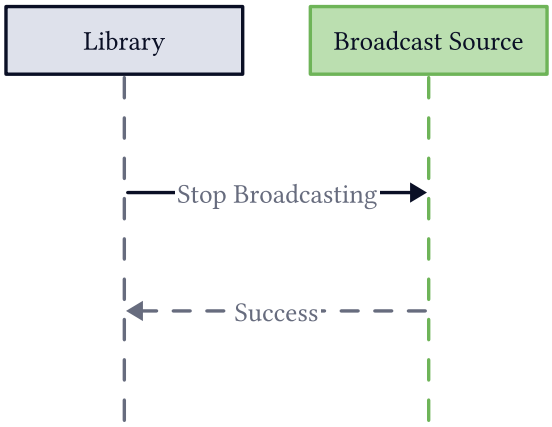

Figure 19: Library design - Stopping the broadcast

5.4.4. Connecting to the Broadcast Sink

To be able to connect to a broadcast sink the assistant needs to scan for available devices first. The assistant will respond with the found devices, which the library then receives. Found found devices will be sent constantly while scanning for broadcast sinks until the stop command is sent. After choosing the device to which the application wants to connect with the library sends the chosen device back to the assistant which is then connecting to it. When a selection is made the library sends a stop scanning command so the assistant is not scanning forever. In the end, a connect command will be sent to the assistant which is then establishing the connection to the sink.

Figure 20 visualizes the steps needed to establish the broadcast sink connection.

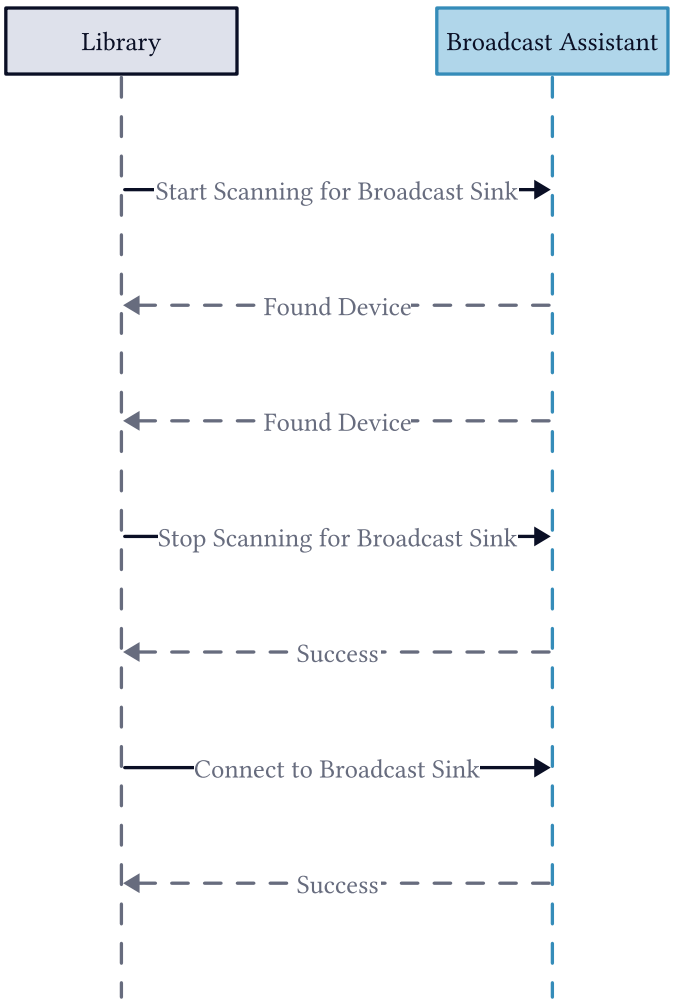

Figure 20: Library design - Connecting to a broadcast sink

5.4.5. Selecting the Broadcast

When the connection to the broadcast sink is established a broadcast needs to be selected which the sink device then can listen to. For scanning the available broadcasts the sequence is similar to the one searching for broadcast sinks *Section 5.4.4, Connecting to the Broadcast Sink*. The assistant will need to discover available broadcasts from the broadcast source and then send them back to the library. When selecting a broadcast the assistant then needs to tell the connected broadcast sink to which broadcast it should listen to.

Figure 21 visualizes the steps needed to select a broadcast.

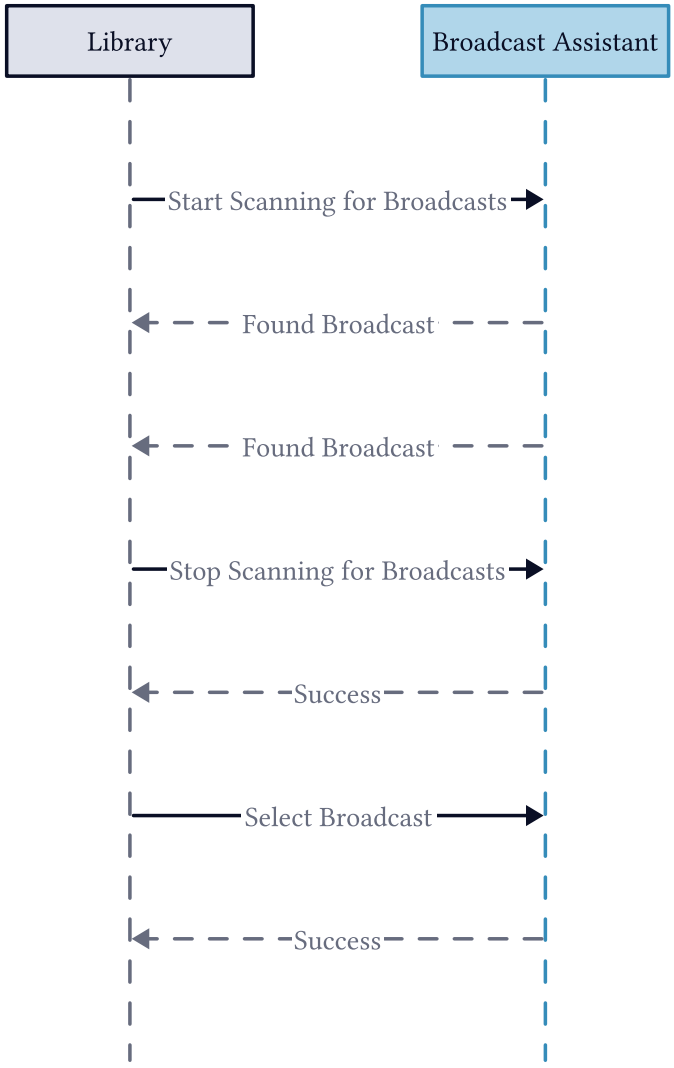

Figure 21: Library design - Selecting a broadcast

5.5. Components

To visualize the components used in our solution, which we chose to implement, we created a component diagram shown in Figure 22. Double-ended arrows represent actions that have an associated response, whereas normal arrows indicate fire-and-forget actions.

On the right side, the two development kits are represented. The top one is the broadcast source, and the bottom one is the broadcast assistant. Both contain the various services that we defined and implemented. The services present in both embedded apps are not duplicated; instead, they come from a shared module.

On the left side, the PC running the test can be seen with our library encapsulated within. Below the PC is the actual test target, which is an Aurcast receiver.

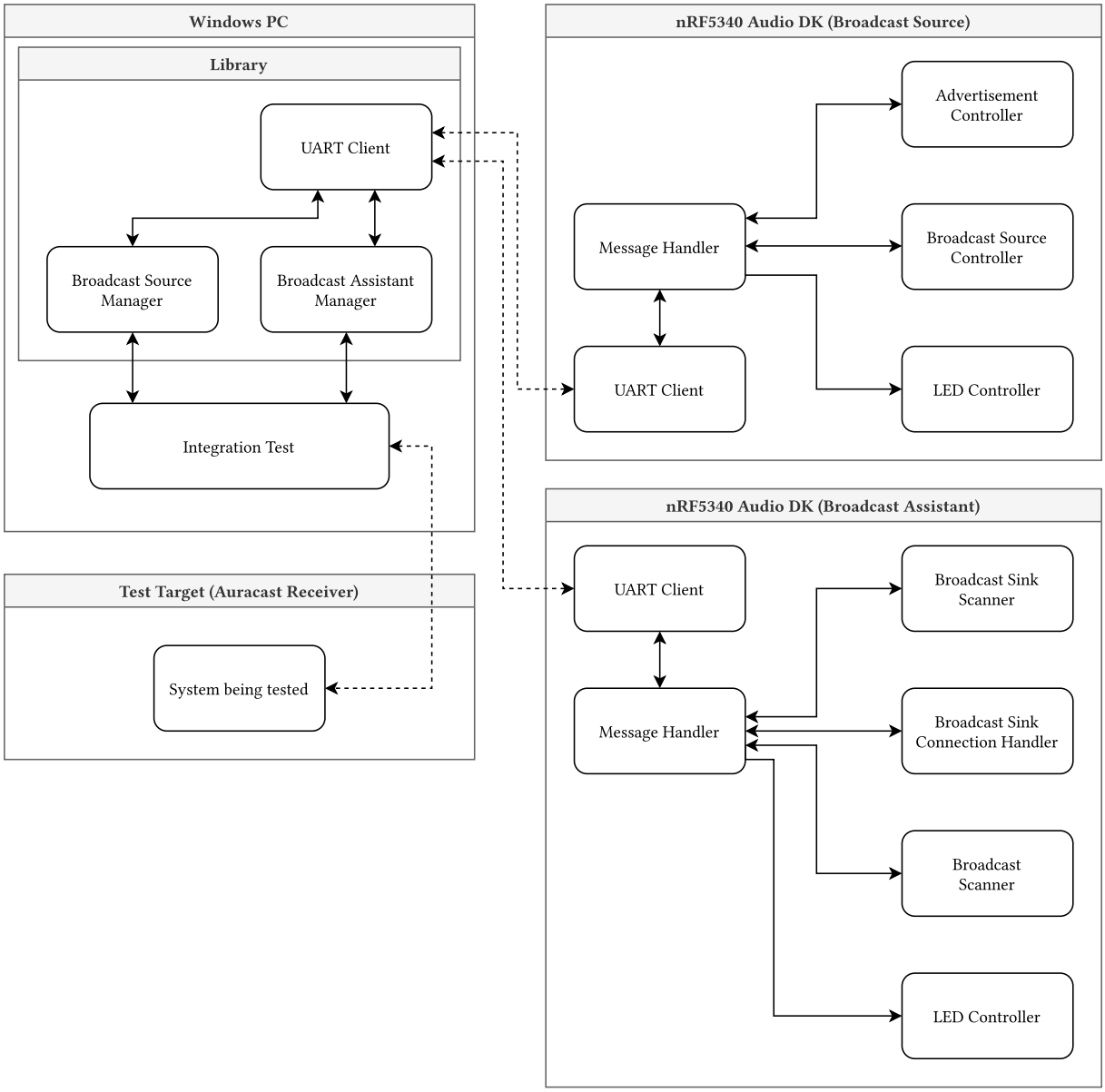

Figure 22: Component diagram

6. Development Process

Within this section, we explain how the working environment was set up and what tools and extensions we used. *Section 6.1, Repository Setup* documents how the repository was set up. In *Section 6.2, Development Environment* we describe how we set up our development environment. What libraries we used is documented in *Section 6.3, Libraries*. Finally, in *Section 6.4, Quality Measures* we outlined how we ensured the quality of our code.

6.1. Repository Setup

For this project, we decided to work with a monolithic repository (Monorepo) to keep the overhead low and allow for faster iterations. This also allows us to share the protocol buffer definitions without having to have them in a separate repository and using something like git submodules [32].

This source repository, along with the one for our documentation, is located on a Sonova-owned GitHub Enterprise instance.

The detailed directory structure that we used can be found in *Section 7, Implementation*.

6.2. Development Environment

We were each provided with a user account and a laptop, which we used for all development tasks. We also had the possibility of working at the Sonova office, where we had access to test devices, such as broadcast source dongles, mobile phones, and headphones.

For development we used Visual Studio Code (VS Code) [33] and VS Codium [34] with various language server extensions to support the development process. The extensions we used can be found in Table 33.

A full integrated development environment could unfortunately not be used as the licensing of our educational versions forbid us from using them for commercial uses. Getting a license from Sonova would have been possible, but not trivial.

Language	**Extension**			**Language Server**		
C++	clangd	0.1.28	[35]	clangd	18.1.2	[36]
C#	C#	2.24.17	[37]	bundled		
Typst	Typst LSP	0.13.0	[38]	bundled		

Table 33: Used VS Code extensions

6.3. Libraries

During development we used various libraries to achieve our goals. The ones for .NET are described in *Section 6.3.1, .NET* and the ones used for the embedded applications in *Section 6.3.2, Embedded.*

6.3.1. .NET

Within the .NET projects we used NuGet [39] packages to include external tools. Table 34 lists all libraries we used within our .NET projects, including their license.

Name	Version	License
CommandLineParser [40]	2.9.1	MIT License
Google.Protobuf [30]	3.26.1	Copyright 2008 Google Inc. All rights reserved. (Listing 19)
Grpc.Tools [41]	2.62.0	Apache License 2.0
NLog [42]	5.2.8	BSD-3-Clause License
StyleCop.Analyzers [43]	1.1.118	MIT License
System.IO.Ports [44]	8.0.0	MIT License

Table 34: Used external libraries in .NET projects

6.3.2. Embedded

We only used the Zephyr operating system and the library functions it exposes. Table 35 shows detailed information about it, including its license.

Name	Version	License
Zephyr [20]	3.6.0	Apache License 2.0

Table 35: Used external libraries in embedded projects

6.4. Quality Measures

Code quality is a crucial aspect of software development. Defining guidelines helps to keep the code consistent, making the code easier to comprehend. Code reviews and Pull Requests (PRs) help to ensure that the code is correct and helps to share knowledge between developers.

6.4.1. Pull Requests

Documentation and code should never be pushed directly to the main branch. A mandatory code review by the other team member is part of every PR. When reviewing a PR, the reviewer should check for quality and consistency in a number of different areas.

- **Easy to read and understand**
  This includes using clear and descriptive names for variables, functions, and classes, avoiding overly complex expressions or statements, and using consistent formatting and indentation.
- **Following standards**
  Code should follow established coding standards and style guides recommended by the programming language community or organization. This includes adhering to guidelines for naming conventions and commonly used programming patterns.
- **Maintainability**
  Code should be written with maintainability in mind, allowing easy modifications and extension without introducing unexpected errors. Overly complex structures should be avoided and suitable design patterns are to be used.
- **Functionality**
  The changes should work as intended and discussed in meetings.
- **Documentation**
  Documentation within the code should only exist if it helps to make the code easier to understand.

6.4.2. Code Quality

To ensure that our code is in good shape and follows state of the art guidelines we were using tools to help us check this.

For the C# projects (Library and Command Line Interface (CLI)) we used an EditorConfig [45] to maintain consistent code. As coding guidelines for C# we used the settings from roslyn [46].

VS Code does not support EditorConfig naively, so a plugin is required. Following the recommendations from Microsoft we used the EditorConfig for VS Code plugin [47].

To be able to also check more detailed code formatting, we decided to use StyleCopAnalyzers [43]. Unfortunately some rules can be defined within EditorConfig and StyleCopAnalyzers resulting in conflicts. To solve this issue we deactivated the rules from StyleCopAnalyzers causing the conflicts.

For the C++ code for the embedded part we followed the coding guidelines defined by Sonova. This includes a clang format configuration, which enforces a certain formatting style, as well as a wiki detailing things like naming conventions and design guidelines. We also had compiler warnings enabled with the default configuration provided by Zephyr.

6.4.3. Continuous Integration

To aid in reviewing PRs, it would have been nice to have something like GitHub Actions [48] or Jenkins [49] to give some basic level of assurance. This would have brought us quite a bit more assurance that the code formatting was correct and that the main branch always builds. It would also have allowed us to build our documentation continuously, and an up-to-date PDF version would always have been available.

Unfortunately, this infrastructure was not available to us, as our industry partner does not use GitHub Actions, and creating a Jenkins setup would have been too complex, given the relatively short duration of our project.

7. Implementation

The simplest way to understand our setup is to take a look at the top-level directory structure of the repository. This can be found in Listing 2.

Each of the top-level directories is described in detail in the following sections. The contents of the `dotnet` directory, containing the library, command-line interface, and sample applications, are described in *Section 7.2, .NET*. The contents of the `embedded` directory, containing the two embedded apps, are described in *Section 7.1, Embedded Applications*. Finally, the contents of the `protos` directory, containing the protocol buffer definitions, are described in *Section 7.4, Serial Protocol*.

```
├── dotnet
│   ├── cli
│   ├── library
│   └── samples
├── embedded
│   └── apps
└── protos
```

Listing 2: Directory structure of repository

7.1. Embedded Applications

The embedded applications were implemented in C++ and are located within the `embedded` directory, whose structure is visualized in Listing 3. Each application has its own source directory with shared code being located in the `shared` directory.

The implementation details of the broadcast source are described in *Section 7.1.1, Broadcast Source* and the ones of the broadcast assistant is described in *Section 7.1.2, Broadcast Assistant*. Finally, the shared code is looked at in *Section 7.1.3, Shared Code*.

```
embedded
└── apps
    ├── broadcast-assistant
    ├── broadcast-source
    └── shared
```

Listing 3: Directory structure of embedded applications

7.1.1. Broadcast Source

The broadcast source application has been adapted from a Zephyr example application, which was written in C. We tried to extract its functionality into sensible classes. The resulting structure can be seen in Listing 4.

Most of the classes fall into the categories of "advertisement related" or "broadcast source related" and thus are located in appropriate directories. Table 36 contains a short description of each class's functionality.

```
broadcast-source
├── CMakeLists.txt
└── src
    ├── Advertisement
    │   ├── AdvertisementConfiguration.hpp
    │   ├── AdvertisementController.hpp
    │   └── AdvertisementWrapper.hpp
    ├── BroadcastSource
    │   ├── BroadcastSourceConfiguration.hpp
    │   ├── BroadcastSourceController.hpp
    │   ├── BroadcastSourceWrapper.cpp
    │   └── BroadcastSourceWrapper.hpp
    ├── Logger.hpp
    ├── main.cpp
    └── ProtobufTraits.hpp
```

Listing 4: Directory structure of the broadcast source application

Class	Description
`AdvertisementConfiguration`	Holds configuration values for the `AdvertisementWrapper`.
`AdvertisementController`	Responsible for managing an instance of an `AdvertisementWrapper`.
`AdvertisementWrapper`	Takes an `AdvertisementConfiguration` as a constructor argument and then starts advertising. Stops advertising upon destruction.
`BroadcastSourceConfiguration`	Holds configuration values for the `BroadcastSourceWrapper`.
`BroadcastSourceController`	Responsible for managing an instance of a `BroadcastSourceWrapper`.
`BroadcastSourceWrapper`	Takes a `BroadcastSourceConfiguration` as a constructor argument, starts broadcasting once given an advertisement to use. Stops broadcasting upon destruction.
`Logger`	Contains an alias for the `GenericLogger` (from the shared directory) with the broadcast source specific log message.
`ProtobufTraits`	Contains trait definitions used by the `ProtobufEncoder` and `ProtobufDecoder` (from the shared directory).

Table 36: Description of the broadcast source application classes

7.1.2. Broadcast Assistant

The broadcast assistant application has been adapted from a Zephyr example application, which was written in C. We tried to extract its functionality into sensible classes. The resulting structure can be seen in Listing 5.

Table 37 contains a short description of each class's functionality.

```
broadcast-assistant
├── CMakeLists.txt
└── src
    ├── BroadcastSink
    │   ├── BroadcastSinkConnectionHandler.hpp
    │   └── BroadcastSinkScanner.hpp
    ├── BroadcastSource
    │   └── BroadcastScanner.hpp
    ├── Logger.hpp
    ├── main.cpp
    └── ProtobufTraits.hpp
```

Listing 5: Directory structure of the broadcast assistant application

Class	Description
`BroadcastSinkConnectionHandler`	Handles all interaction with the broadcast sink, including connection establishment and sending it the chosen broadcast id.
`BroadcastSinkScanner`	Scans for available broadcast sinks, checks each sink's Auracast™ capabilities, and sends the ones that do support Auracast™ to the library.
`BroadcastScanner`	Scans for available broadcasts and sends them to the library.
`Logger`	Contains an alias for the `GenericLogger` (from the shared directory) with the broadcast assistant specific log message.
`ProtobufTraits`	Contains trait definitions used by the `ProtobufEncoder` and `ProtobufDecoder` (from the shared directory).

Table 37: Description of the broadcast assistant application classes

7.1.3. Shared Code

Several pieces of code were written for reuse across both applications. They have been placed in the `shared` directory, whose contents are depicted in Listing 6. Each of those classes is described in Table 38.

```
shared
├── GenericLogger.hpp
├── LedController.hpp
├── ProtobufDecoder.hpp
├── ProtobufEncoder.hpp
├── UartClient.hpp
├── VarintDecoder.hpp
└── VarintEncoder.hpp
```

Listing 6: Contents of the Shared Embedded Directory

File	Description
`GenericLogger`	The logger used to send log messages from the embedded applications to the library. It is generic, because in order for it to encode messages it needs to know the type of the enclosing Protobuf message, which is provided through a template argument by each application.
`LedController`	Handles `SetColor` messages and sets the appropriate General-Purpose Input/Output (GPIO) pins.
`ProtobufEncoder`	Encodes Protobuf messages into a byte array.
`ProtobufDecoder`	Decodes byte arrays into Protobuf messages.
`UartClient`	Handles all serial interaction with the library by registering itself for the appropriate interrupts. It dispatches incoming messages on a separate thread to the provided given callback. Uses the `VarintEncoder` and `VarintDecoder` to read and write message sizes.
`VarintEncoder`	Encodes ints into byte arrays. GPIO
`VarintEncoder`	Decodes byte arrays into ints.

Table 38: Description of the shared embedded classes

7.2. .NET

This section explores the implementation of the library in *Section 7.2.1, Library* and the implementation of the command-line application in *Section 7.2.2, CLI Application*.

7.2.1. Library

The library handles all communication between the two boards and the test running. It also relays log messages from the embedded devices to the chosen .NET logging framework. We chose an abstraction level that we thought was appropriate and ended up with two exposed classes, `BroadcastSourceManager` and `BroadcastAssistantManager`. For filtering found devices we also exposed two record types used in predicates, `BroadcastSink` and `Broadcast`. They can be seen in the class diagram in Figure 23.

The differentiation into the two manager classes makes it transparent to the consumer with which board they are interacting with. Since there is quite a bit of shared logic they both inherit from the abstract `BaseManager`. They also both take a serial number during construction, which they use to target the correct board. All method calls are blocking operations and, with the exception of the `SetColor` command, wait for the board response before returning.

The library is also responsible for forwarding log messages from the board to the logging system that we use, along with their importance level. This means that the global log-level set also affects whether log events from the embedded boards are logged. However, the log level on the embedded boards does not get changed, the library will just silently drop messages below the importance threshold.

The `BroadcastSourceManager` exposes four public methods, which are described in Table 39. The `BroadcastAssistantManager` exposes four public methods, which are described in Table 40.

Method	Description
`SetColor`	1. Sends the `SetColor` command.
`FollowLogMessages`	1. Waits for `LogMessage` messages from the board and logs them.
`StartBroadcasting`	1. Sends the `StartBroadcasting` command. 2. Waits for the `StartBroadcasting` response message.
`StopBroadcasting`	1. Sends the `StopBroadcasting` command. 2. Waits for the `StopBroadcasting` response message.

Table 39: Public methods of the broadcast source manager

Method	Description
`SetColor`	1. Sends the `SetColor` command to the board.
`FollowLogMessages`	1. Waits for `LogMessage` messages from the board and logs them.
`ConnectToBroadcastSink`	1. Sends the `ScanForBroadcastSink` command. 2. Waits for `ScanForBroadcastSink` response message. 3. Waits for `ScanResult` messages until a match is found. 4. Sends `StopScanningForBroadcastSink` command. 5. Waits for `StopScanningForBroadcastSink` response message. 6. Sends `ConnectToBroadcastSink` command. 7. Waits for `ConnectToBroadcastSink` response message.
`SelectBroadcastSource`	1. Sends `ScanForBroadcastSource` command. 2. Waits for `ScanForBroadcastSource` response message. 3. Waits for `ScanResult` messages until a match is found. 4. Sends `StopScanningForBroadcastSource` command. 5. Waits for `StopScanningForBroadcastSource` response message. 6. Sends `SelectBroadcastSink` command. 1. Waits for `SelectBroadcastSink` response message.

Table 40: Public methods of the broadcast assistant manager

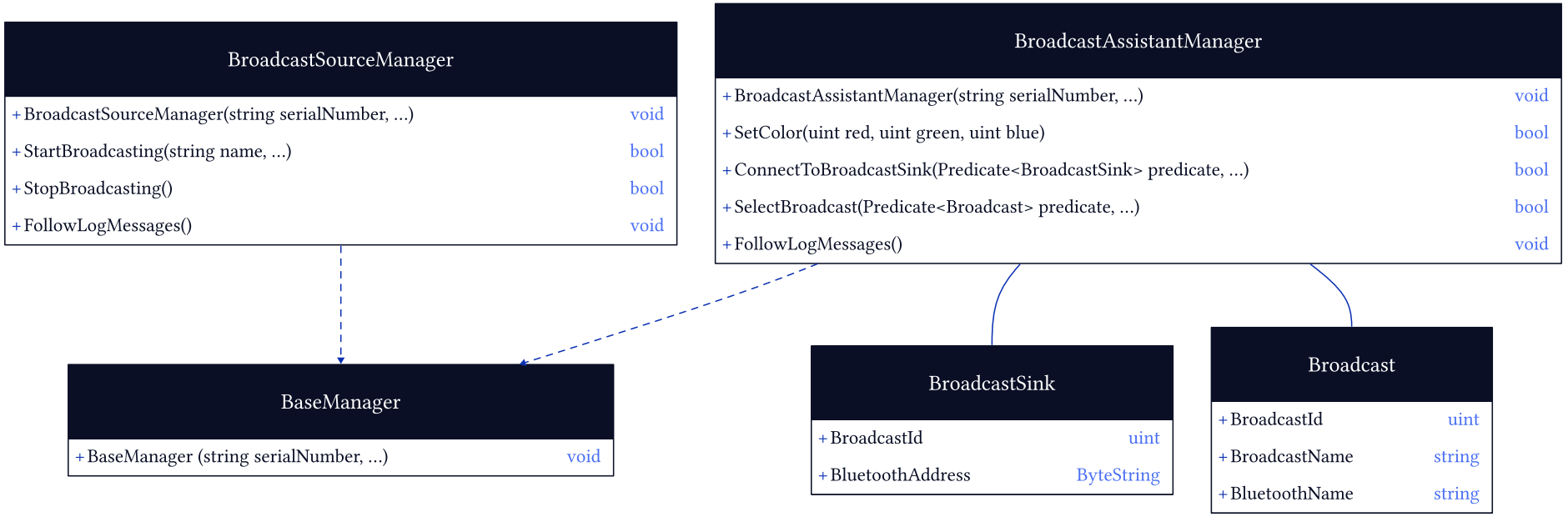

Figure 23: Library class diagram showing public classes, methods and fields

7.2.2. CLI Application

To test the library we implemented a simple CLI application allowing us to use all public functions of the library described in Table 39 and Table 40.

We created four `Option` classes which define the parameters used to execute the library functions. Parameters to set the board color and to connect to the serial port are used in multiple options, to reuse them we created interfaces and collected them in the `Interfaces` directory. The `OptionParser.cs` uses the Command Line Parser library [40] to parse the options and then calls the appropriate library functions.

The resulting file structure can be seen in Listing 7.

```
cli
├── Program.cs
├── OptionParser.cs
└── Options
    ├── Interfaces
    │   ├── IColorOptions.cs
    │   ├── IPortOptions.cs
    ├── BroadcastSourceOptions.cs
    ├── ColorOptions.cs
    ├── ConnectToBroadcastSinkOptions.cs
    ├── PrintOptions.cs
    └── SelectBroadcastOptions.cs
```

Listing 7: Directory structure of the CLI application

7.2.2.1. Usage

Before connecting to a specific serial port we need to know which ports are available. To find available Nordic boards, the command in Listing 8 be used. It prints the serial number and port number of each device found.

```
dotnet run print --connected-devices
```

Listing 8: CLI command to print connected devices

It is important that the serial number of the correct board with the needed application is used. For `set-color --sink`, `connect-sink` and `select-broadcast` the broadcast sink application described in *Section 7.1.2, Broadcast Assistant* needs to be flashed onto the board. For `set-color --source`, `start-broadcast` and `stop-broadcast` the broadcast assistant application described in *Section 7.1.1, Broadcast Source* is needed.

Setting the LED color

To test if the setup works, the color of the LED can be set. This can be done with the `set-color` command, whose flags are documented in Table 41. A sample invocation can be found in Listing 9.

Long	Short	Description
`--source`	`-s`	Sets the color on the broadcast source device
`--assistant`	`-a`	Sets the color on the broadcast assistant device
`--serial-number`	`-`	Serial number or the Nordic board
`--red`	`-r`	Sets the color red (0 or 255, optional)
`--green`	`-g`	Sets the color green (0 or 255, optional)
`--blue`	`-b`	Sets the color blue (0 or 255, optional)

Table 41: Flags for the `set-color` command

```
dotnet run set-color --serial-number 1050156338 -g 255
```

Listing 9: Example of the `set-color` command

Starting a Broadcast

A broadcast has to be started first, before a sink can listen to it. To do this the `start-broadcasting` command can be used, by specifying the serial number and defining the name of the broadcast. The available flags for the command are listed in Table 42 and an example invocation in Listing 10.

Long	Short	Description
`--serial-number`	`-`	Serial number or the Nordic board
`--follow`	`-f`	Continues printing log messages after the command is executed until interrupted
`--name`	`-n`	Name of the broadcast which is being started

Table 42: Flags for the `start-broadcasting` command

```
dotnet run start-broadcasting --serial-number 1050156338 --name Test-Broadcast
```

Listing 10: Example of the `start-broadcasting` command

Connecting to a Broadcast Sink

To connect a broadcast sink, like ear buds, the `connect-sink` command needs to be executed. The broadcast sinks can be filtered by Bluetooth® name and Bluetooth® address. The first broadcast sink matching the filters is selected and at least one filter needs to be set. The available flags for the command are listed in Table 43 and an example invocation in Listing 11.

Long	Description
`--serial-number`	Serial number or the Nordic board
`--bluetooth-name`	Filters the available broadcast sinks by their Bluetooth® name
`--bluetooth-address`	Filters the available broadcast sinks by their Bluetooth® address

Table 43: Flags for the `connect-sink` command

```
dotnet run connect-sink --serial-number 1050164593 --bluetooth-name 'Galaxy Buds2 White'
```

Listing 11: Example of the `connect-sink` command

Selecting a Broadcast

After establishing a connection to the the broadcast sink, a broadcast can be selected. This is also done on the broadcast assistant, using the `select-broadcast` command. The broadcasts can be filtered by broadcast id, broadcast name, and Bluetooth® name. The first broadcast matching the filters is selected and at least one filter needs to be set. The available flags for the command are listed in Table 44 and an example invocation in Listing 12.

Long	Description
`--serial-number`	Serial number or the Nordic board
`--broadcast-id`	Filters the available broadcasts by the broadcast id
`--broadcast-name`	Filters the available broadcasts by the broadcast name
`--bluetooth-name`	Filters the available broadcasts by the Bluetooth® name

Table 44: Flags for the `select-broadcast` command

```
dotnet run select-broadcast --serial-number 1050164593 --broadcast-name Test-Broadcast
```

Listing 12: Example of the `select-broadcast` command

Common Flags

Additionally you can follow the log messages until they are interrupted with `Ctrl. + C`. This works with `start-broadcast`, `connect-sink` and `select-broadcast`. The flag is shown in Table 45.

Long	Short	Description
`--follow`	`-f`	Continues printing log messages after the command is executed until interrupted

Table 45: Common flags for the CLI application

7.3. Serial Communication

In *Section 5.3, Serial Protocol* we chose to use Protocol Buffers as the message format. This brings with it an additional challenge, because, unlike strings, Protocol Buffers do not specify a termination character. This means we need to find a different way to find the start and end of messages.

We chose to implement a header in the form of a variable sized integer, which specifies the size of the following message. A fixed-size header would also have been sufficient, but to be as future-proof as possible we decided to go with the variable sized approach. Figure 24 shows how the approach looks like with "Length" being the variable sized integer and the "Message" being the encoded Protocol Buffer message.

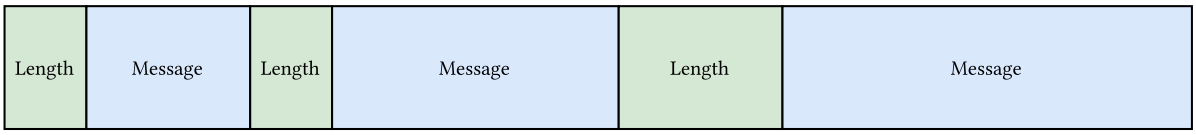

Figure 24: Illustration of the serial communication

The implementation we chose for the variable-sized integers uses the first bit of a byte to indicate whether more length-indicating bytes are following or not. The other seven bits following the indication bit are used to actually encode the length. An example of this process is shown in Figure 25, where two bytes are used to encode a message length of 2766 bytes. The first byte has the continuation bit (shown in green) set to one, indicating that there is more to follow. The second byte has the continuation bit set to zero, indicating that it is the final byte to consider to determine the length.

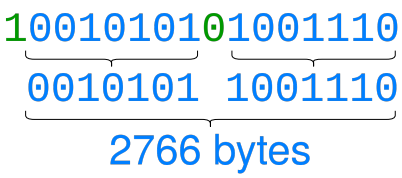

Figure 25: Example decoding of a variable sized integer

7.4. Serial Protocol

When using Protocol Buffers the protocol format needs to be defined in `.proto` files. In addition a `.option` file can be used to set specific requirements for fields for example setting a fixed length for a string. These files can then be used by all applications involved. In our case we added the files to the `protos` directory, which is also shown in Listing 2. This directory is then used by the library and embedded applications.

Listing 13 shows the `.proto` file for shared commands and Listing 14 show the one for shared messages.

In general we divided the Protobuf messages into commands and messages to distinguish them. A command is only sent from the library to the boards and a message is only sent from the boards to the library. A message can contain a command response, a log message, or a scan result. As the message is not necessarily a response we decided to not name it `Response`.

Each time a command is sent a response message is expected, except for the `SetColor` command, which is mainly used for debugging purposes. The response message contains a boolean field `success` indicating whether the command was successful or not. Until the response message arrives multiple `LogMessages` can be expected.

```
message SetColor {
    // values between 0 and 255
    uint32 red = 1;
    uint32 green = 2;
    uint32 blue = 3;
}
```

Listing 13: Protocol definition - Shared commands

```
message LogMessage {
    enum Level {
        Debug = 0;
        Info = 1;
        Warning = 2;
        Error = 3;
    }

    Level level = 1;
    string message = 2;
}
```

Listing 14: Protocol definition - Shared messages

7.4.1. Broadcast Source

The broadcast source can receive three different commands `SetColor`, `StartBroadcasting` and `StopBroadcasting` which are defined in the `BroadcastSourceCommand.proto` which is shown in Listing 15.

All response and log messages are defined in the `BroadcastSourceMessage.proto` file shown in Listing 16. The used response messages for the broadcast source are `StartBroadcastingResponse` and `StopBroadcastingResponse`. The response messages are set as `CommandResponse` in the `BroadcastSourceMessage`.

```
import "CommonCommands.proto";

message BroadcastSourceCommand {
  oneof command {
    SetColor setColor = 1;
    StartBroadcasting startBroadcasting = 2;
    StopBroadcasting stopBroadcasting = 3;
  }

  message StartBroadcasting {
    string name = 1;
    CodecConfiguration codecConfiguration = 2;

    message CodecConfiguration {
      Lc3Config lc3Config = 1;
      QosParameters qosParameters = 3;

      enum SamplingFrequency {
        INVALID = 0;
        KHZ_8 = 1;
        KHZ_11 = 2;
        ...
        KHZ_192 = 12;
        KHZ_384 = 13;
      }

      enum FrameDuration {
        MSEC_7_5 = 0;
        MSEC_10 = 1;
      }

      message Lc3Config {
        SamplingFrequency samplingFrequency =
1;
        FrameDuration frameDuration = 2;
        uint32 octetsPerFrame = 3;
      }

      message QosParameters {
        uint32 maxSduSize = 1;
        uint32 maxLatency = 2;
        uint32 frameInterval = 3;
        uint32 presentationDelay = 4;
      }
    };
  }

  message StopBroadcasting { }
}
```

Listing 15: Protocol definition - Broadcast source commands

```
import "CommonMessages.proto";

message BroadcastSourceMessage {
  oneof message {
    LogMessage logMessage = 1;
    CommandResponse commandResponse = 2;
  }

 message CommandResponse {
    oneof response {
      StartBroadcastingResponse
startBroadcastingResponse = 1;
      StopBroadcastingResponse
stopBroadcastingResponse = 2;
    }

    message StartBroadcastingResponse {
      bool success = 1;
    }

    message StopBroadcastingResponse {
      bool success = 1;
    }
  }
}
```

Listing 16: Protocol definition - Broadcast source messages

7.4.1.1. Set Color

To set the color of the LED on the board the `SetColor` command is used, which is defined within the shared `.proto` file shown in Listing 13. The process is visualized in Figure 26.

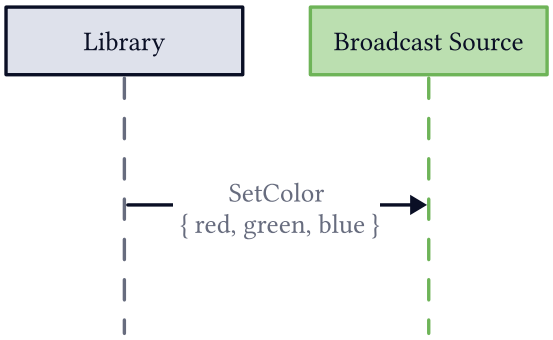

Figure 26: Implementation - Setting the LED color on the broadcast source

7.4.1.2. Start Broadcasting

The `StartBroadcasting` command contains a parameter to set the name of the broadcast and configurations for the codec. Details of the parameters can be found in the `.proto` file in Listing 15.

As a response the `StartBroadcasting` message is sent back to the library. Details of the response message can be found in the `.proto` file in Listing 16.

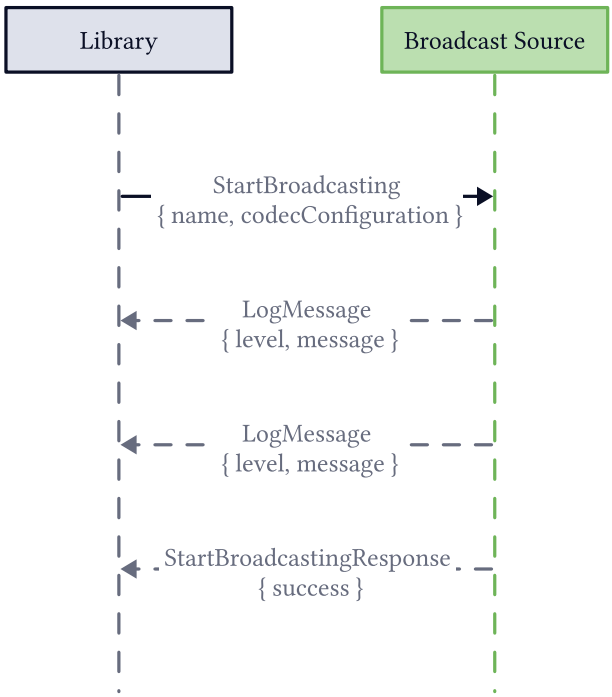

Figure 27: Implementation - Starting the broadcast

7.4.1.3. Stop Broadcasting

The `StopBroadcasting` command contains no parameters. Details of the command can be found in the `.proto` file in Listing 15.

As a response the `StopBroadcasting` message is sent back to the library. Details of the response message can be found in the `.proto` file in Listing 16.

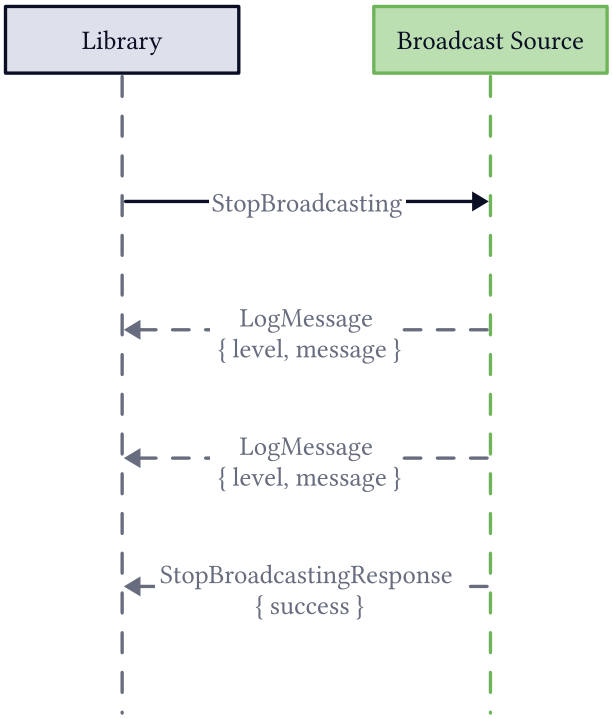

Figure 28: Implementation - Stopping the broadcast

7.4.2. Broadcast Assistant

The broadcast assistant can receive seven commands: `SetColor`, `StartScanningForBroadcastSink`, `StartScanningForBroadcastSource`, `StopScanningForBroadcastSink`, `StopScanningForBroadcastSource`, `ConnectToBroadcastSink` and `ConnectToBroadcastSource`. They are defined in `BroadcastAssistantCommand.proto`, which is shown in Listing 17.

All response and log messages are defined in the `BroadcastAssistantMessage.proto` file shown in Listing 18. The used response messages for the broadcast assistant are `StartScanningForBroadcastSinkResponse`, `StartScanningForBroadcastSourceResponse`, `StopScanningForBroadcastSinkResponse`, `StopScanningForBroadcastSourceResponse`, `ConnectToBroadcastSinkResponse` and `ConnectToBroadcastSourceResponse`. The response messages are nested under `CommandResponse` in the `BroadcastAssistantMessage`.

```
import "CommonCommands.proto";

message BroadcastAssistantCommand {
  oneof command {
    SetColor setColor = 1;
    StartScanningForBroadcastSink startScanningForBroadcastSink = 2;
    StopScanningForBroadcastSink stopScanningForBroadcastSink = 3;
    ConnectToBroadcastSink connectToBroadcastSink = 4;
    StartScanningForBroadcast startScanningForBroadcast = 5;
    StopScanningForBroadcast stopScanningForBroadcast = 6;
    SelectBroadcast selectBroadcast = 7;
  }

  message StartScanningForBroadcastSink {}
  message StopScanningForBroadcastSink {}
  message StartScanningForBroadcast {}
  message StopScanningForBroadcast {}
  message ConnectToBroadcastSink {
    bytes bluetoothAddress = 1;
    uint32 type = 2;
  }
  message SelectBroadcast {
    uint32 broadcastId = 1;
  }
}
```

Listing 17: Protocol definition - Broadcast assistant commands

```
import "CommonMessages.proto";

message BroadcastAssistantMessage {
  oneof message {
    LogMessage logMessage = 1;
    CommandResponse commandResponse = 2;
    ScanResult scanResult = 3;
  }

  message CommandResponse {
    oneof response {
      StartScanningForBroadcastSinkResponse
          startScanningForBroadcastSinkResponse = 1;
      StopScanningForBroadcastSinkResponse
          stopScanningForBroadcastSinkResponse = 2;
      ConnectToBroadcastSinkResponse connectToBroadcastSinkResponse = 3;
      StartScanningForBroadcastResponse startScanningForBroadcastResponse = 4;
      StopScanningForBroadcastResponse stopScanningForBroadcastResponse = 5;
      SelectBroadcastResponse selectBroadcastResponse = 6;
    }

    message StartScanningForBroadcastSinkResponse { bool success = 1; }
    message StopScanningForBroadcastSinkResponse { bool success = 1; }
    message ConnectToBroadcastSinkResponse { bool success = 1; }
    message StartScanningForBroadcastResponse { bool success = 1; }
    message StopScanningForBroadcastResponse { bool success = 1; }
    message SelectBroadcastResponse { bool success = 1; }
  }

  message ScanResult {
    oneof result {
      BroadcastSinkScanResult broadcastSinkScanResult = 1;
      BroadcastScanResult broadcastScanResult = 2;
    }

    message BroadcastSinkScanResult {
      string bluetoothName = 1;
      uint32 type = 2;
      bytes bluetoothAddress = 3;
    }

    message BroadcastScanResult {
      string bluetoothName = 1;
      string broadcastName = 2;
      uint32 broadcastId = 3;
    }
  }
}
```

Listing 18: Protocol definition - Broadcast assistant messages

7.4.2.1. Set Color

To set the color of the LED on the assistant board the `SetColor` command is used, which is defined within the shared `.proto` file shown in Listing 13. The process is visualized in Figure 26.

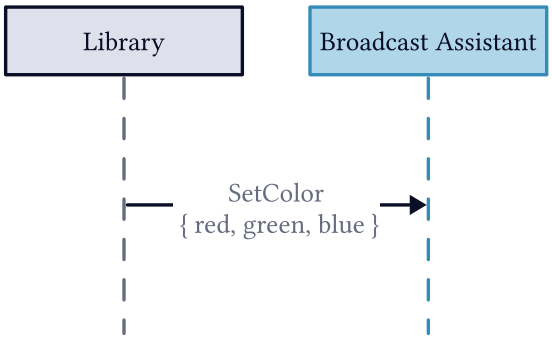

Figure 29: Implementation - Setting the LED color on the broadcast assistant

7.4.2.2. Connect to Broadcast Sink

To establish a connection to the broadcast sink a series of commands should be sent.

The `StartScanningForBroadcastSink` command starts scanning for broadcast sinks and sends a `BroadcastSinkScanResult` message when a potential device has been found.

When a suitable broadcast sink is found, the scanning needs to be stopped by sending the `StopScanningForBroadcastSink` command.

Then the `ConnectToBroadcastSink` command is used to establish the connection. The command contains multiple parameters to set the address and the type of the broadcast sink to which the board should connect to.

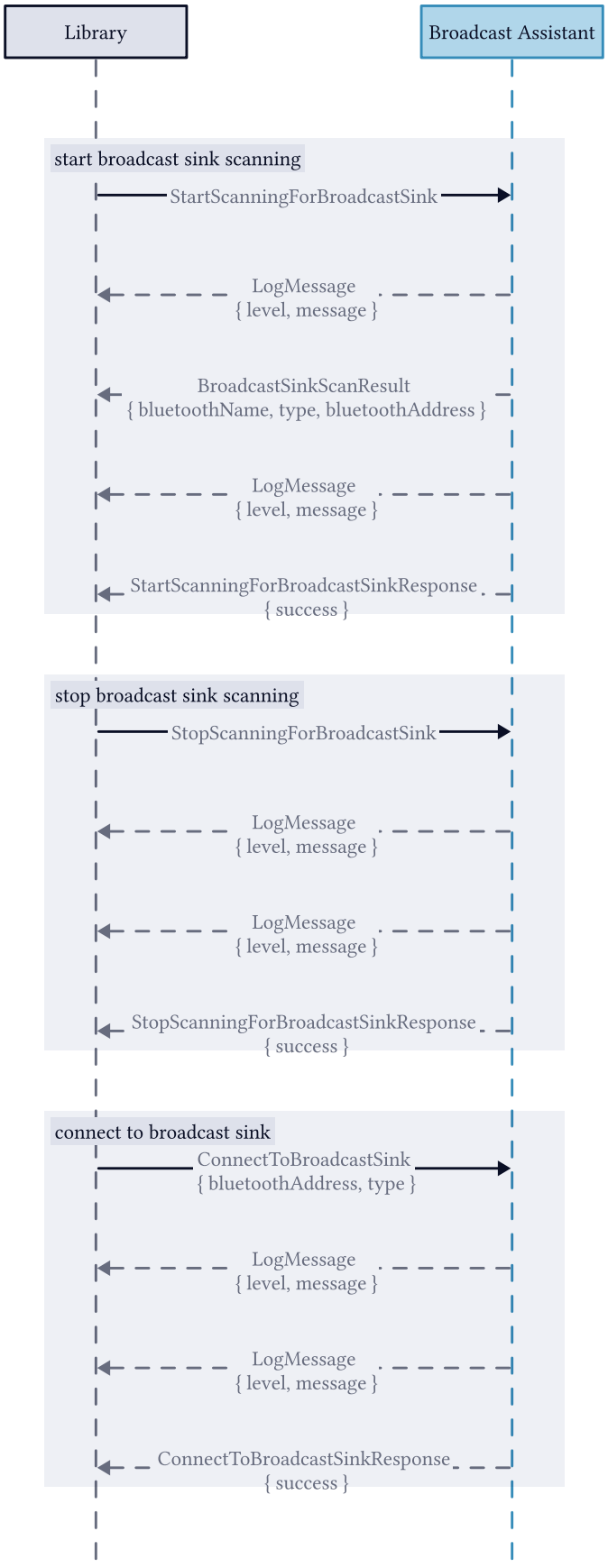

Figure 30: Implementation - Connecting the broadcast sink

7.4.2.3. Select Broadcast

To select a broadcast a series of commands should be sent.

The `StartScanningForBroadcast` command starts scanning for available broadcasts and responds with `BroadcastScanResult` when an available broadcast has been found.

When a suitable broadcast is found, the scanning needs to be stopped by sending the `StopScanningForBroadcast` command.

Then the `SelectBroadcast` is used to select a broadcast, meaning the connected broadcast sink will be instructed to listen to it. The command contains only one parameter to set the broadcast id of the chosen broadcast.

Details of the commands and messages can be found in the `.proto` files in Listing 17 and Listing 18.

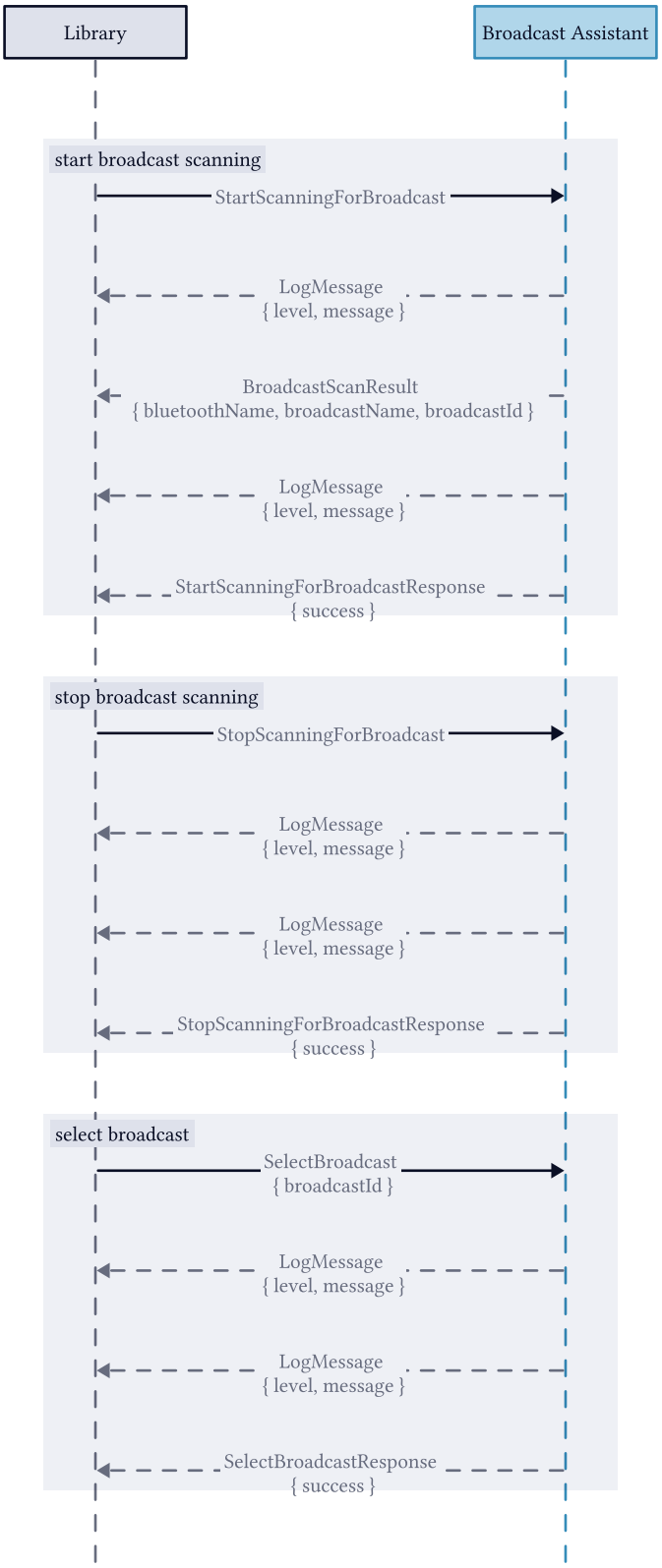

Figure 31: Implementation - Select a broadcast

8. Verification

To ensure the quality of our solution, this section details the verification process.

A review of our requirements (defined in *Section 3, Requirements*) is performed in *Section 8.1, Functional Requirements Review* and in *Section 8.2, Non-Functional Requirements Review*.

Our approach to automated testing is described in *Section 8.3, Automated Testing* and the one for manual testing in *Section 8.4, Manual Testing*.

8.1. Functional Requirements Review

In this section we take a look at the requirements defined in *Section 3.1, Functional Requirements*. For each defined requirement we determined whether it is fulfilled or not.

8.1.1. Broadcast Source

All MVPs were achieved. However, the more advanced features were not implemented due to time constraints. See Table 46 for details.

Requirement	Status	Comment
FR101 - Starting and Stopping the Stream **MVP**	✓	-
FR102 - Multiple Streams	✗	This has not been implemented due to time constraints. However, this functionality can be achieved by using multiple boards simultaneously.
FR103 - Naming the Streams	✓	-
FR104 - Playing Test Tone **MVP**	✓	-
FR105 - Audio Input Selection	✗	This has not been implemented due to time constraints.
FR106 - Encrypted Streams	✗	This has not been implemented due to time constraints.

Table 46: Broadcast source - Requirement review

8.1.2. Broadcast Assistant

All requirements were achieved. See Table 47 for details.

Requirement	Status	Comment
FR201 - Scanning for Sink Devices **MVP**	✓	-
FR202 - Pairing of Specific Device **MVP**	✓	-
FR203 - Connecting to Specific Device **MVP**	✓	-
FR204 - Selecting Stream and Sending to Sink **MVP**	✓	-
FR205 - Scanning for Broadcast Sources	✓	-
FR206 - Scanning for Broadcast Sinks	✓	-

Table 47: Broadcast assistant - Requirement review

8.1.3. Library

Unfortunately the functional requirements for the library were not implemented, due to time constraints. However, they were not part of the MVP. See Table 48 for details.

Requirement	Status	Comment
FR301 - Receiving GATT Information	✗	This has not been implemented due to time constraints.

Table 48: Library - Requirement review

8.2. Non-Functional Requirements Review

In this section we take a look at the requirements defined in *Section 3.2, Non-Functional Requirements*. For each defined requirement we determined whether it is fulfilled or not.

All defined requirements, even non-MVP ones, were achieved.

8.2.1. Functionality

All functionality-related requirements were achieved. Table 49 lists all requirements and their verification status.

Requirement	Status	Comment
NFR101 - Comply with Industry Standards **MVP**	✓	-
NFR102 - Dependency Management **MVP**	✓	-

Table 49: Functionality - Requirement review

8.2.2. Usability

All usability-related requirements were achieved. Table 50 lists all requirements and their verification status.

Requirement	Status	Comment
NFR201 - Graceful Error Handling **MVP**	✓	-
NFR202 - Abstraction Level **MVP**	✓	-
NFR203 - Ease of Use **MVP**	✓	-
NFR204 - Usage Examples **MVP**	✓	-

Table 50: Usability - Requirement review

8.2.3. Reliability

All reliability-related requirements were achieved. Table 51 lists all requirements and their verification status.

Requirement	Status	Comment
NFR301 - Reliable Operation	✓	-

Table 51: Reliability - Requirement review

8.2.4. Performance

All performance-related requirements were achieved. Table 52 lists all requirements and their verification status.

Requirement	Status	Comment
NFR401 - Startup Time **MVP**	✓	-
NFR402 - Performance of Multiple Streams	✓	-

Table 52: Performance - Requirement review

8.2.5. Supportability

All supportability-related requirements were achieved. Table 53 lists all requirements and their verification status.

Requirement	Status	Comment
NFR501 - Integration of Library **MVP**	✓	-
NFR502 - Maintainability **MVP**	✓	-
NFR503 - Clear Versioning System	✓	-
NFR504 - Platform Agnostic Implementation	✓	-

Table 53: Supportability - Requirement review

8.3. Automated Testing

Automated unit tests were written for the C# library. They are located in the `Tests` subdirectory of the `library` directory and test all public functionality of the `BroadcastSourceManager` and `BroadcastAssistantManager` classes to verify that the correct messages are sent to the board.

8.4. Manual Testing

During development each public method was tested manually to verify it is working correctly. This included using test devices, like the Samsung Galaxy Buds 2 Pro, and running the applications on the DKs. For these tests we also used the CLI heavily. Additional hardware was used as needed, such as a phone supporting Auracast™, like the Galaxy S23 and S24. To aid in debugging also used the Nordic Wireless nRF Connect for Mobile app to scan for available bluetooth devices. [50]

Manual Testing was performed before opening a PR and many times also by the person reviewing the PR.

9. Challenges

In this chapter we documented some of the challenges we encountered during development. This section is split into an embedded (*Section 9.1, Embedded*) and a dotnet part (*Section 9.2, Dotnet*).

9.1. Embedded

During embedded development, we encountered the most difficulties, as at the beginning of this project, we were both quite inexperienced in this area. Many of them stem from the issue that the Zephyr SDK is a C library, which often takes pointers to some data. This makes it quite difficult to keep track of the data we need to keep intact.

9.1.1. Work Queues and Bluetooth® Library

While implementing UART communication (*Section 5.3, Serial Protocol*), some Bluetooth® functionality seemingly randomly stopped working. We first suspected our interrupt-based messaging handling, since we processed everything inside the interrupt handler. We thought that maybe it would help if we switched to a polling-based approach. This did indeed work. However, constantly polling for new bytes is not the best solution either.

We then read about Zephyr's threading system, and discovered workqueue threads [51]. There already is a system workqueue in place and according to Zephyr's documentation

> Additional workqueues should only be defined when it is not possible to submit new work items to the system workqueue.
>
> — Zephyr [51]

So we submitted work items to the system workqueue and expected the issue to be solved; however, the device still halted.

As a last-ditch effort, we created our own workqueue, which ultimately solved the issue. We learned that the Bluetooth® library uses the system workqueue and does not create its own. Other developers have also reported this issue, and the misleading documentation [52], [53].

9.1.2. Configurable Stack Size

During the development of the broadcast source, the board seemed to halt unexpectedly during a call to a Zephyr library function. To investigate this, we added print statements to both ours and Zephyr's code. We identified the specific print statements between which the board halted and attempted to diagnose the issue. Everything seemed to work as expected; no errors were detected before the halt.

We discussed this issue with our advisor, and one of the first questions he asked was if the stack size is configurable. This was the case, and after increasing the configured stack size, the issue with the unexpected halt was solved.

The root cause here was the switch from the system workqueue to our own (*Section 9.1.1, Work Queues and Bluetooth® Library*). Within our own workqueue, we used the default stack size, which was a lot smaller than the system workqueue that we used before.

9.1.3. Pointers to Nowhere

When implementing the advertising features of the broadcast source, we were using a mobile phone to scan for the Auracast™ stream. The stream never appeared, and we were out of ideas, so we asked a member of Sonova's embedded software team for advice. Together, we took a traffic capture using an Ellisys analysis system [54].

The advertising data looked very suspicious, similar to uninitialized memory. This gave us the clue we needed, and we checked the code for places where we pass pointers to temporary data. We found a few of them and refactored the entire broadcast source controller to prevent the issue from happening again.

9.2. Dotnet

During the development of the .NET project, we encountered fewer problems than with the embedded ones. However, we still had some challenges that kept us working.

9.2.1. Serial Ports

When connecting to a serial port to communicate with the Nordic board, the COM port is needed. We wanted to print available ports in the CLI, but it turned out to be more complex than we thought. After hours of research, we still did not manage to find the right way to read out the serial number of the Nordic boards so the user could simply enter the serial number instead of the COM port. In the end, we simply used nrfjprog, one of the command line tools from Nordic [55]. This is not the best solution, since it just calls an external executable, but it seems to work fine and does what we need it to.

9.2.2. Development Environments

Unfortunately, we could not use development environments like Rider or Visual Studio as described in *Section 6.2, Development Environment*. The result of only using VS Code was that many programming errors were not spotted and code completion did not work at times. It took us some time to get used to it, but in the end, it worked out fine. Dealing with these issues made the implementation time of features a bit longer than it would have with fully integrated development environments.

10. Project Management

The approach of the project is explained in *Section 10.1, Approach*. The planning of the project is looked into in *Section 10.2, Project Plan*, including a comparative analysis between the initial plan and its effective implementation. Finally, *Section 10.3, Time Tracking* provides a summarized overview of the allocated working hours coupled with a reflective assessment of the time invested.

10.1. Approach

To discuss the ongoing process of the project we had a weekly meeting with our advisor, where we discussed what work has been done and what is planned for the next week. Receiving feedback on the ongoing development is quite important, as the direction we are going towards can be assessed and adjusted in just a week.

The development is managed using the GitHub instance of Sonova as described in *Section 6, Development Process*. For a better overview of what is already done, PRs are used to review each other's changes. This same method is also used for writing the documentation, which allows tracking all changes easily.

For the documentation, Typst [56] was used, which is a new markup-based typesetting system. In VS Code, there is an extension supporting Typst [38], which includes a version of the Typst compiler.

To keep track of tasks we needed to do, we used GitHub issues [57], which we updated weekly.

The project is split into three main parts.

Analysis and Experimentation The analysis consisted of research on what Auracast™ is and how it works. This step also entailed gathering knowledge needed for the implementation, such as looking into the Bluetooth® standard. We experimented with the Nordic boards we received, and tried out the ready-to-use sample applications. In addition, we also looked into already existing code from another team at Sonova to check if we can use or extend it for our solution. The experimentation phase also included looking for possible solutions, including proof of concepts, which can be presented to the Sonova team.

Project Setup After the decision was made on which direction the project should go, we set up the project. In this step, we installed all the needed tools and created the repositories on the Sonova GitHub instance.

Implementation and Finalization In the implementation phase, the embedded applications, CLI, and serial protocol were implemented according to *Section 5, Design*. We made sure the code follows the Sonova guidelines and is well structured.

10.2. Project Plan

Our project plan is shown in Figure 32. The darker hues represent the plan we made initially and the lighter hues show the actual time spent.

Initially we stayed pretty close to the plan, especially in the analysis phase. However, during this phase we also learned that the test system integration would not be our responsibility, but would be done by someone else afterwards. This led to the “Test System Integration” part not being done at all, which gave us some much needed time for the implementation and documentation and in hindsight would have been a bit too much given the limited time we had.

10.2.1. Important Deadlines

To help us stay on top of important appointments and deadlines, we have compiled them in this section. This will ensure we have a clear overview and can manage our time effectively.

25. March, 13:00	Intermediate presentation
10. June	Hand in AVT abstract
12. June, 18:00	Poster hand-in
13. June, 12:00	Print documentation
14. June, 16:00	Start of exhibition
14. June, 17:00	Final hand-in
24. June, 13:00	Final presentation

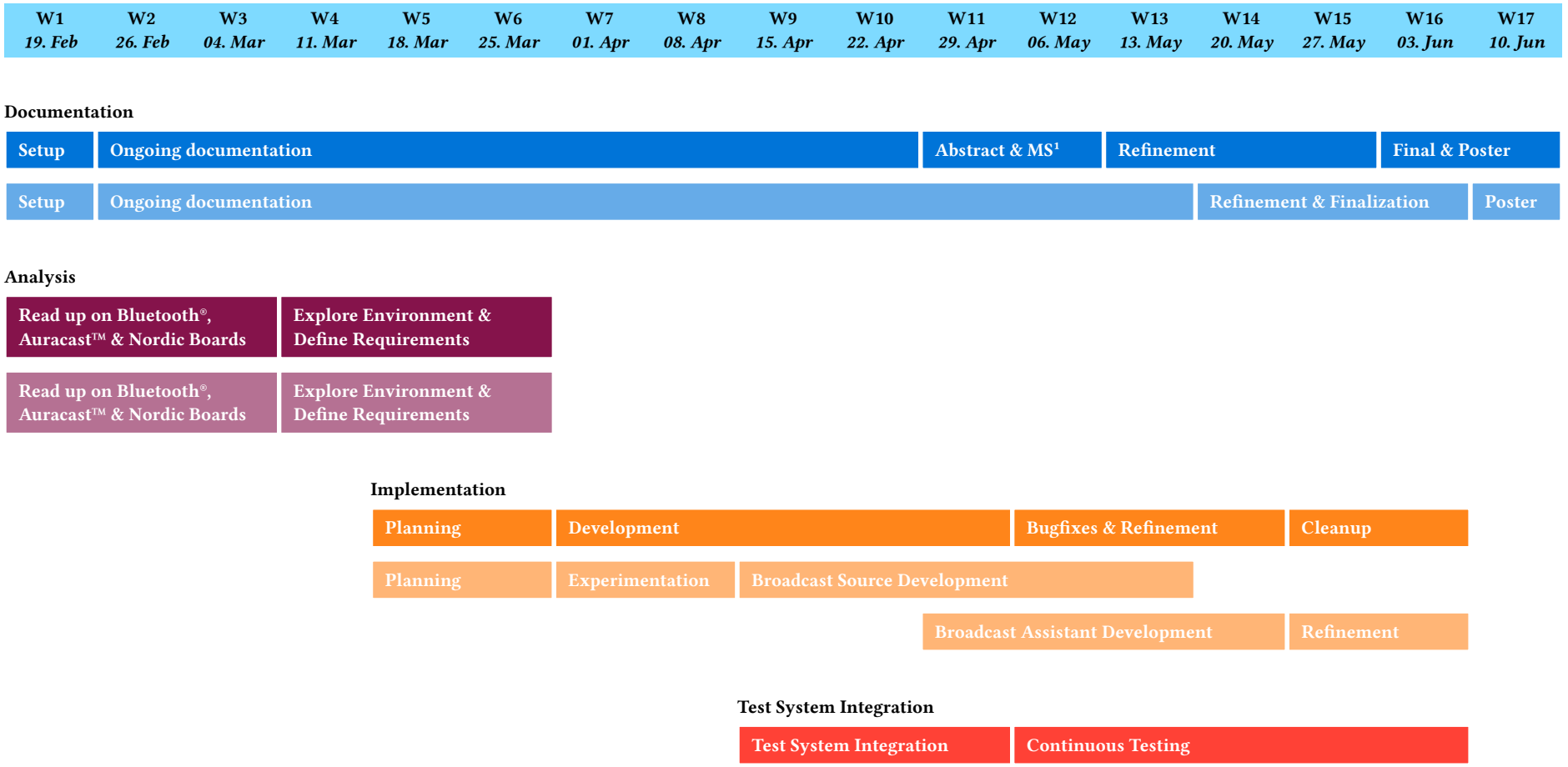

Figure 32: Project plan

[1]Management Summary

10.3. Time Tracking

To monitor the working hours effectively, a Google Sheet was established, where information about the time spent was meticulously recorded. Each record contains a date, the amount of time spent, name of executor, task category and a brief comment detailing the specific work performed during that time. Table 54 and Figure 33 show the total amount of time spent on each category per project week. Figure 34 shows the share of time invested per category. Most of the time spent was invested into the documentation and implementation, with the former being the main focus at the end of the project. Figure 35 shows the time spent by each project author. Both authors contributed a similar amount to both documentation and implementation.

SUM of Hours	*Type*						
Week	Admin	Documentation	Experimentation	Implementation	Meeting	Studying / Research	Grand Total
Week 07	1.5h				1.0h		2.5h
Week 08	14.0h	5.0h	16.0h			2.0h	37.0h
Week 09	4.0h	5.5h			3.0h	31.5h	44.0h
Week 10	5.0h	6.0h	12.5h		3.6h	19.5h	46.6h
Week 11	3.5h	23.5h	6.5h	1.0h	6.0h	6.5h	47.0h
Week 12	7.8h	13.5h	3.8h		7.6h	11.0h	43.6h
Week 13	3.5h	15.5h	2.0h	10.0h	5.0h	1.0h	37.0h
Week 14	1.3h	5.8h		27.5h	4.0h		38.5h
Week 15		2.5h		40.0h	3.6h	0.3h	46.3h
Week 16	2.5h	3.5h		31.3h	3.0h	1.5h	41.8h
Week 17	1.3h	2.0h		31.0h	7.0h	2.0h	43.3h
Week 18	0.3h	2.0h		26.1h	5.6h	1.0h	34.9h
Week 19	0.5h	1.0h		40.0h	3.0h		44.5h
Week 20		4.5h		32.5h	3.5h		40.5h
Week 21		18.5h		29.0h	2.0h		49.5h
Week 22		32.5h		20.5h	5.0h		58.0h
Week 23	1.5h	49.5h		12.0h	5.0h		68.0h
Week 24	21.0h	43.5h		3.0h	2.0h		69.5h
Week 25	40.0h						40.0h
Week 26					8.0h		8.0h
Grand Total	**107.6h**	**234.3h**	**40.8h**	**303.9h**	**77.7h**	**76.3h**	**840.4h**

Table 54: Time invested per category and project week

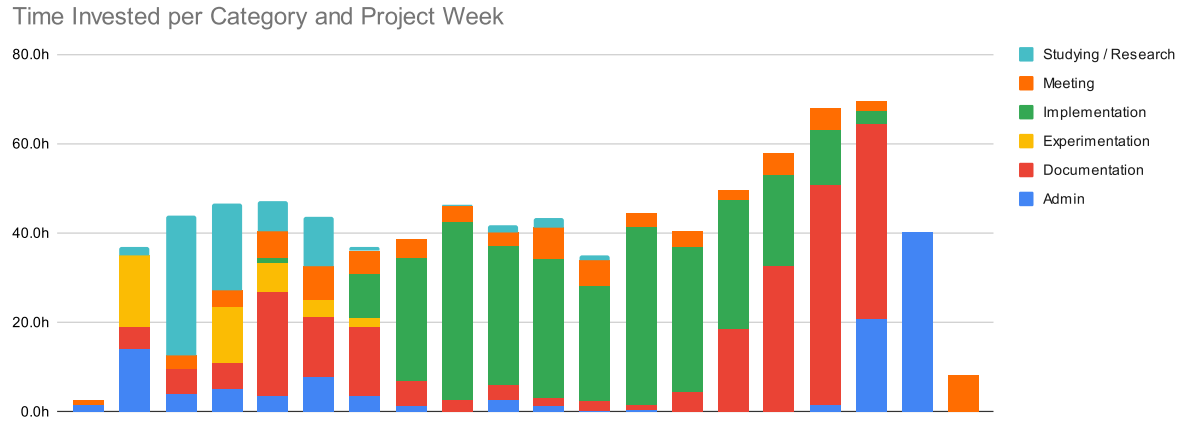

Figure 33: Time invested per category and project week

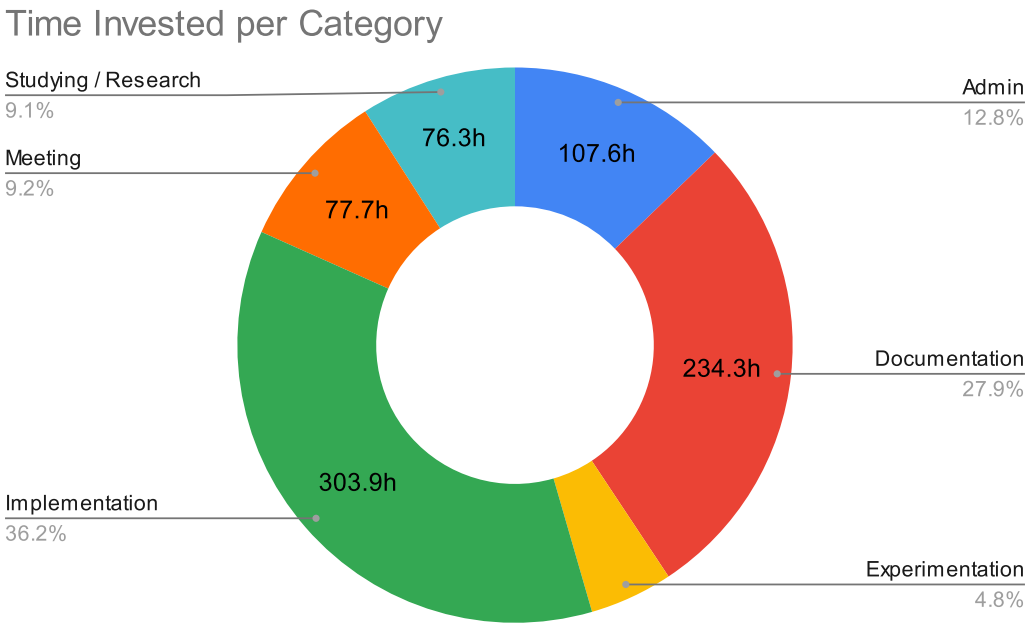

Figure 34: Time invested per category

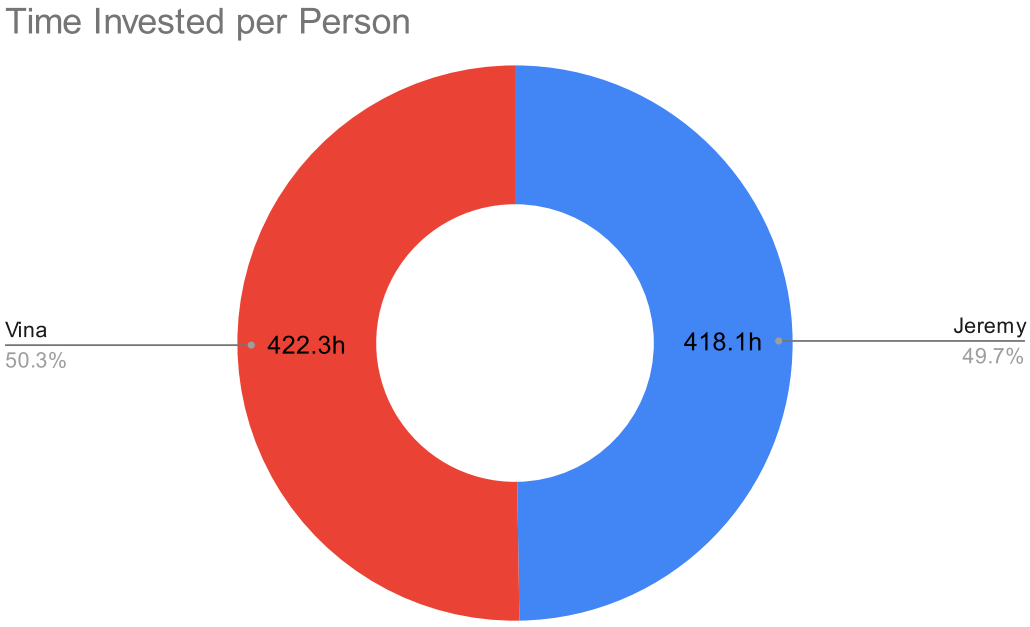

Figure 35: Time invested per person

11. Conclusion

In this thesis, we analyzed BLE and Bluetooth® Auracast™, where we learned a lot about the inner workings of Bluetooth®. This included reading about ATT, GATT, CIS, CIG, and many more. We focused our detailed analysis mainly on Auracast™.

As a next step, we came up with ideas on how a test system could be structured. We compared different approaches to the overall architecture and later also compared different options for the serial communication between the library and the boards.

After a decision for the approach was reached, we started the implementation process. We successfully implemented a .NET library, a CLI (also written in .NET), and two embedded applications, one for the broadcast source and one for the broadcast assistant.

Our implemented solution fulfills all MVPs, however some other requirements were not implemented due to time constraints.

We will go over our learnings during the project in *Section 11.1, Learnings* and take a look at its future in *Section 11.2, Outlook*.

11.1. Learnings

One of the biggest learning experiences was coming up with the architecture of our solution. For both of us, it was the first time we had the chance to do so, and we learned a lot. We came up with different ideas, prepared the options for a pitch meeting, and discussed them with experts.

We also learned a lot about embedded development and C++. Especially the pitfalls of interacting with a C library provided lots of learning opportunities for us. The many challenges we encountered during embedded development are documented in *Section 9.1, Embedded*.

Working with an industry partner was also very insightful, as we got an insider look at the company. Experiencing how embedded software is being developed up close and the challenges the team encountered was very interesting.

Working on a closed-source project also posed challenges like licensing restrictions, which we were not particularly used to. We had to look at the licensing of each tool and library we wanted to use much closer than we usually do.

11.2. Outlook

After the end of our thesis, there will be a handover process, where we formally pass the project on to Sonova. They will most likely make some adjustments to the project to make it fit for integration into their test system, such as creating a NuGet package and updating their internal documentation.

What we built will aid their embedded development team in building products with Auracast™ support. It would also be easily adaptable to any future test system, even if it is not .NET, since the library is not that big and can be rewritten in any language, without having to adjust the protocol definitions or embedded applications.

We look forward to seeing devices with Auracast™ support from Sonova in the future and hope that our contribution will make a difference.

12. Disclaimers

Parts of this paper were rephrased using GPT-3.5 [58], GPT-4 [59], and GPT-4o [60]. This paper was reviewed by our advisor and corrections were applied according to the comments made. For spell checking and translations DeepL [61], LanguageTool [62], grammarly [63], and QuillBot [64] were used.

13. Glossary

***ATT* – Attribute Protocol**: See *Section 2.4.1, Attribute Protocol (ATT) - Roles*. 12, 13, 79, 87

***BASS* – Broadcast Audio Scan Service**: A protocol that allows Bluetooth® devices to scan for and receive audio broadcasts from a source device without requiring a prior pairing, enabling seamless audio streaming to multiple devices simultaneously. 2

***BLE* – Bluetooth® Low Energy**: See *Section 2.1, Bluetooth® LE (BLE)*. 4, 5, 7, 11, 12, 79, 87

***CIG* – Connected Isochronous Groups**: Can support multi-connected data streaming with one master and multiple slaves. Each group can contain multiple CIS instances. 6, 79

***CIS* – Connected Isochronous Streams**: Is a logical transport that enables connected devices to transfer isochronous data unidirectionally and bidirectionally. 6, 79, 81

***CLI* – Command Line Interface**: A text-based user interface used for entering commands to a computer program. Users interact with software or operating systems by typing commands into a terminal or console window. 43, 52, 53, 55, 73, 74, 88, 90, 93

***COM* – Component Object Model**: Interface exposed by Windows to interact with other processes in an abstract way. In the context of this thesis we used it to communicate with the development kit via a serial connection over USB. 28, 73

***DK* – Development Kit**: Special hardware with extra debugging functionality 2, 3, 4, 11, 24, 71

***FURPS* – Functionality, Usability, Reliability, Performance and Supportability**: Serves as a framework for evaluating and defining the non-functional requirements of a software system. 18

***GAP* – Generic Access Profile**: See *Section 2.1.1, Generic Access Profile (GAP)*. 5, 12, 88

***GATT* – Generic Attribute Profile**: See *Section 2.4, Bluetooth® Generic Attribute Profile (GATT)*. 12, 16, 17, 79

***GPIO* – General-Purpose Input/Output**: A versatile pin on an integrated circuit used for digital signaling, capable of being configured as either an input or an output. In embedded systems, GPIOs are essential for interfacing with various peripherals and sensors. 48

***HCI* – Host Controller Interface**: Provides standardized communication between a host and a Bluetooth® controller. Usually both are on the same device or chip, but it can also be used across other connections like USB. 26, 29, 32, 88

***HFP* – Hands-Free Profile**: The Hands-Free Profile specification defines the interactions for hands free devices, for example cellular phones with a Bluetooth® in-car kit. This profile provides means for both remote control of the device and voice connections. 6

***JSON* – JavaScript Object Notation**: A language-independent data interchange format. 31, 33, 34, 35

***Monorepo* – monolithic repository**: A single version control repository that houses multiple projects or components, simplifying dependency management, promoting code reuse, and enhancing collaboration. 41

***MVP* – Minimum Viable Product**: A version of a product that includes only the essential features necessary to provide it to users and gather initial feedback before making further developments and improvements. 14, 68, 69, 79

***PBP* – Public Broadcast Profile**: Defines how a broadcast source can use extended advertising data (AD) to signal that it is transmitting broadcast audio streams that can be discovered and rendered by broadcast sinks that support commonly used audio configurations. 8

***PR* – Pull Request**: A method used in version control systems to submit contributions to a project by requesting that changes from one branch be merged into another. In other environments, such as GitLab [65], it is also known as a merge request. 43, 71, 74

***RF* – Radio Frequency**: Portion of the electromagnetic spectrum from 3 kHz to 300 GHz, used for transmitting data wirelessly through the air. 5

***SCO* – Synchronous Connection-Oriented**: Is one of the two possible Bluetooth® data link types defined. It is a symmetric, point-to-point link between the master device and a specific slave device. 6

***SDK* – Software Development Kit**: A collection of software tools, libraries, and documentation provided to developers to create applications for a specific platform or framework. It simplifies development by offering pre-built functions and interfaces. 11, 28, 72

***UART* – Universal Asynchronous Receiver-Transmitter**: A hardware communication protocol that handles asynchronous serial communication between devices. In the context of this thesis we used it to communicate over the serial connection between the development kit and the computer." 27, 28, 32, 72, 87, 90

***UI* – User Interface**: The visual and interactive aspect of a software application or device, encompassing all elements that users interact with, such as buttons, menus, and icons, to facilitate user interaction and enhance user experience. 31

***UUID* – Universally Unique Identifier**: A universally unique identifier is a specific form of identifier which can be safely deemed unique for most practical purposes. Two correctly generated UUIDs have a virtually negligible chance of being identical, even if they're created in two different environments by separate parties. This is why UUIDs are said to be universally unique. 13

***VS Code* – Visual Studio Code**: Visual Studio Code is a streamlined code editor with support for development operations like debugging, task running, and version control. It aims to provide just the tools a developer needs for a quick code-build-debug cycle and leaves more complex workflows to fuller featured IDEs. 41, 43, 73, 74, 88

14. Bibliography

[1] "For Assistive Listening." Accessed: Jun. 10, 2024. [Online]. Available: https://www.bluetooth.com/auracast/assistive-listening

[2] "This is Sonova." Accessed: Mar. 28, 2024. [Online]. Available: https://www.sonova.com/en/sonova-0

[3] "The next generation of Bluetooth® audio." Accessed: Mar. 28, 2024. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/feature-enhancements/le-audio/

[4] SAMSUNG, "Experience Immersive, Intelligent Sound With Galaxy Buds Series' Latest Updates." Accessed: Mar. 18, 2024. [Online]. Available: https://news.samsung.com/global/experience-immersive-intelligent-sound-with-galaxy-buds-series-latest-updates

[5] "nRF5340 Audio DK." Accessed: Mar. 01, 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF5340-Audio-DK

[6] "The Difference Between Classic Bluetooth® and Bluetooth® Low Energy." Accessed: Jun. 09, 2024. [Online]. Available: https://blog.nordicsemi.com/getconnected/the-difference-between-classic-bluetooth-and-bluetooth-low-energyr

[7] "Bluetooth® Low Energy (BLE): A Complete Guide." Accessed: Jun. 08, 2024. [Online]. Available: https://novelbits.io/bluetooth-low-energy-ble-complete-guide

[8] "Bluetooth® Wireless Technology." [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/tech-overview

[9] M. Afaneh, "How Bluetooth® Low Energy Works: Advertisements." Accessed: Feb. 29, 2024. [Online]. Available: https://novelbits.io/bluetooth-low-energy-advertisements-part-1

[10] M. Afaneh, "Mastering BLE: A Guide to Peripherals and Centrals." Accessed: Jun. 08, 2024. [Online]. Available: https://novelbits.io/ble-peripherals-centrals-guide

[11] "LE Audio: The Future of Bluetooth® Audio." Accessed: Jun. 09, 2024. [Online]. Available: https://www.bluetooth.com/wp-content/uploads/2022/10/MRN-LE_Audio.pdf

[12] "The next generation of Bluetooth® audio." Accessed: Feb. 22, 2024. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/feature-enhancements/le-audio

[13] M. Afaneh, "The Ultimate Guide to What's New in Bluetooth® version 5.2." Accessed: Jun. 09, 2024. [Online]. Available: https://novelbits.io/bluetooth-version-5-2-le-audio

[14] "How It Works." Accessed: Feb. 22, 2024. [Online]. Available: https://www.bluetooth.com/auracast/how-it-works/

[15] "Developing Auracast™ Receivers with an Assistant Application for Legacy Smartphones." Accessed: Mar. 15, 2024. [Online]. Available: https://www.bluetooth.com/wp-content/uploads/2024/02/2401_Accelerating_Auracast™_Adoption_FINAL.pdf

[16] "Bluetooth® Core Specification Version 5.2 Feature Overview." Accessed: Mar. 18, 2024. [Online]. Available: https://www.bluetooth.com/bluetooth-resources/bluetooth-core-specification-version-5-2-feature-overview

[17] "Galaxy Buds2 Pro." Accessed: Mar. 15, 2024. [Online]. Available: https://www.samsung.com/global/galaxy/galaxy-buds2-pro/

[18] "MOMENTUM True Wireless 4." Accessed: Mar. 15, 2024. [Online]. Available: https://www.sennheiser-hearing.com/en-US/p/momentum-true-wireless-4

[19] “nRF Connect SDK.” Accessed: Feb. 26, 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-software/nRF-Connect-SDK

[20] “The Zephyr Project.” Accessed: Mar. 08, 2024. [Online]. Available: https://www.zephyrproject.org/

[21] K. Townsend, “GATT.” Accessed: Mar. 28, 2024. [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt

[22] “Part G. Generic Attribute Profile (GATT).” Accessed: Jun. 10, 2024. [Online]. Available: https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host/generic-attribute-profile%E2%80%93gatt-.html

[23] M. Afaneh, “Bluetooth® ATT and GATT Explained (Connection-Oriented Communication).” Accessed: Jun. 10, 2024. [Online]. Available: https://novelbits.io/bluetooth-le-att-gatt-explained-connection-oriented-communication

[24] M. Afaneh, “Bluetooth® GATT: How to Design Custom Services & Characteristics [MIDI device use case].” Accessed: Mar. 20, 2024. [Online]. Available: https://novelbits.io/bluetooth-gatt-services-characteristics

[25] “Building and running nRF5340 Audio applications.” Accessed: Feb. 25, 2024. [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/applications/nrf5340_audio/doc/building.html

[26] “Testing and optimization.” Accessed: Feb. 25, 2024. [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/test_and_optimize.html

[27] “Hearing Aid free icon.” Accessed: Feb. 26, 2024. [Online]. Available: https://www.flaticon.com/free-icon/hearing-aid_6431923?term=hearing+aid&page=1&position=6&origin=search&related_id=6431923

[28] “bumble.” Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/google/bumble

[29] “Bluetooth® Profile Drivers.” Accessed: Mar. 15, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/bluetooth

[30] “Protocol Buffers.” Accessed: Apr. 15, 2024. [Online]. Available: https://protobuf.dev/

[31] “Introducing JSON.” Accessed: Apr. 15, 2024. [Online]. Available: https://www.json.org/

[32] “Git Submodules.” Accessed: May 27, 2024. [Online]. Available: https://git-scm.com/book/en/v2/Git-Tools-Submodules

[33] “Visual Studio Code.” Accessed: May 27, 2024. [Online]. Available: https://code.visualstudio.com/

[34] “VSCodium.” Accessed: May 27, 2024. [Online]. Available: https://vscodium.com/

[35] “clangd Visual Studio Code Extension.” Accessed: May 27, 2024. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=llvm-vs-code-extensions.vscode-clangd

[36] “What is clangd?.” Accessed: Jun. 07, 2024. [Online]. Available: https://clangd.llvm.org/

[37] “C# Visual Studio Code Extension.” Accessed: May 27, 2024. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp

[38] “Typst LSP Visual Studio Code Extension.” Accessed: May 27, 2024. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=nvarner.typst-lsp

[39] “An introduction to NuGet.” Accessed: May 02, 2024. [Online]. Available: https://learn.microsoft.com/en-us/nuget/what-is-nuget

[40] “commandline.” Accessed: Feb. 02, 2024. [Online]. Available: https://github.com/commandlineparser/commandline

[41] “grpc.” Accessed: Apr. 25, 2024. [Online]. Available: https://github.com/grpc/grpc

[42] “NLog.” Accessed: Feb. 02, 2024. [Online]. Available: https://github.com/NLog/NLog

[43] “StyleCopAnalyzers.” Accessed: Apr. 15, 2024. [Online]. Available: https://github.com/DotNetAnalyzers/StyleCopAnalyzers

[44] “System.IO.Ports.” Accessed: Apr. 25, 2024. [Online]. Available: https://github.com/nanoframework/System.IO.Ports

[45] “Define consistent coding styles with EditorConfig.” Accessed: Apr. 19, 2024. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/ide/create-portable-custom-editor-options?view=vs-2022

[46] “roslyn/.editorconfig.” Accessed: Apr. 15, 2024. [Online]. Available: https://github.com/dotnet/roslyn/blob/main/.editorconfig

[47] “EditorConfig for VS Code.” Accessed: Apr. 15, 2024. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=EditorConfig.EditorConfig

[48] “GitHub Actions.” Accessed: May 27, 2024. [Online]. Available: https://docs.github.com/actions

[49] “Jenkins.” Accessed: May 27, 2024. [Online]. Available: https://www.jenkins.io/

[50] “nRF Connect for Mobile.” Accessed: Jun. 09, 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-mobile

[51] “Workqueue Threads.” Accessed: May 06, 2024. [Online]. Available: https://docs.zephyrproject.org/latest/kernel/services/threads/workqueue.html

[52] “System workqueue misuse and misleading documentation .” Accessed: May 06, 2024. [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/issues/61819

[53] “Bluetooth®: UART BT using System Workqueue no longer compatible with Blocking BT TX calls.” Accessed: May 06, 2024. [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/issues/72019

[54] “Ellisys Bluetooth® Vanguard Advanced All-in-One Bluetooth® Analysis System.” Accessed: May 06, 2024. [Online]. Available: https://www.ellisys.com/products/bv1/index.php

[55] “nRF Command Line Tools .” Accessed: Jun. 13, 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nrf-command-line-tools

[56] “Typst.” Accessed: Jun. 08, 2024. [Online]. Available: https://typst.app/

[57] “Project planning for developers.” Accessed: Jun. 08, 2024. [Online]. Available: https://github.com/features/issues

[58] “ChatGPT 3.5.” Accessed: May 17, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo

[59] “ChatGPT 4.” Accessed: May 17, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4

[60] “ChatGPT 4o.” Accessed: May 17, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o

[61] “DeepL.” Accessed: Feb. 25, 2024. [Online]. Available: https://www.deepl.com/translator

[62] “LanguageTool.” Accessed: Feb. 25, 2024. [Online]. Available: https://languagetool.org/

[63] “Grammarly.” Accessed: Feb. 25, 2024. [Online]. Available: https://app.grammarly.com/

[64] “QuillBot.” Accessed: Feb. 25, 2024. [Online]. Available: https://quillbot.com/grammar-check

[65] “GitLab.” Accessed: May 27, 2024. [Online]. Available: https://about.gitlab.com/

15. Table of Figures

16. Table of Tables

17. List of Listings

18. Appendix

In this last section the personal reports from both authors can be found in which they reflect on the project (*Section 18.1, Personal Report — Jeremy Stucki* and *Section 18.2, Personal Report — Vina Zahnd*).

The assignment given can be found in *Section 18.3, Assignment*.

Section 18.4, Licenses contains licenses that were referenced.

18.1. Personal Report — Jeremy Stucki

It was very interesting for me to get a look into embedded development. Before this thesis, I only played around with some smaller “Hello World”-type projects, so this was very insightful for me. The most interesting task for me was implementing the serial interaction, including variable-sized int encoding and decoding.

What I found challenging was choosing the right level of abstraction to approach the project, since I knew basically nothing about the Bluetooth® standard and also did not know at which level we would implement our solution.

Writing documentation, as always, was my biggest struggle. I find it very hard to find the correct words and often just want to work on the implementation. Thankfully I was not alone and through our review process we were often able to suggest improvements to each other’s sections.

The working environment was quite different from what I was used to. Using VS Code instead of Rider and CLion was interesting, as I learned a lot about the state of the language servers ecosystem. However, a big challenge for me was using Windows. It has been a number of years since I last used it, and I certainly find it more usable than in the past. For example, package managers like Chocolatey have come a long way. I will still stick with Linux for my personal devices though.

All in all, I am quite happy with the state of the project and am very hopeful that it will be used to test future Auracast™ products. Working on a project with real impact made this thesis very meaningful to me.

I want to thank the entire embedded software team at Sonova for their hospitality, and especially our advisor Thomas Corbat for enabling us to work on this project. Everyone was supporting us when we had questions and we also had some great conversations during lunch or coffee breaks.

18.2. Personal Report — Vina Zahnd

When we first discussed the topic of this project with our advisor, I was glad to hear that the end product is going to be a .NET library, as I also work in this area. But after we started the project, it became more and more clear that we would need to work on the embedded side as well. I was bit overwhelmed by the fact that we had to translate C code into C++ and struggled with it a lot. Unfortunately, when I finally got used to it, we ran out of time to integrate more features.

Then there were also a lot of new Bluetooth® terms to learn, and of course, I would forget the meaning again after a week. I think the problem was that we were working on three different projects at the same time (embedded, library and CLI) and therefore did not go very deep into the Bluetooth® area.

In the end, we did not get as far as we would have liked, as we underestimated the embedded part of the project. It would have been a good project to split into a semester thesis to find multiple solutions for the problem, followed by a bachelor thesis to then implement it.

Having the same project partner for the semester and bachelor thesis was nice, as we already knew how the other person works and what their strengths and weaknesses are. Both of us being busy people, organizing events, going on events, and working at the same time meant we were getting on each other's nerves sometimes. But it never ended in a fight, and we could always discuss problems in a peacefully manner. I am very happy that I could work with Jeremy on this thesis. I learned a lot from him and we also had a good time working together.

At this point, I also want to thank Thomas Corbat for giving us the opportunity to work on this project and supporting us during the whole process. Working at Sonova was inspiring and meeting the team we are doing this project with was very motivating. I hope that our work can be continued by Sonova, and that it was useful to them.

18.3. Assignment

Assignment for Bachelor Thesis "LE Audio Test Infrastructure" Jeremy Stucki and Vina Zahnd

1. Supervisor and Expert

This bachelor Thesis will be developed for the industry partner Sonova AG. It will be supervised by Thomas Corbat (thomas.corbat@ost.ch), OST. An expert independent of OST will examine the thesis and will be present at the final presentation.

- Guido Zgraggen, Google (zgraggen@gmail.com)

2. Students

This project is conducted in the context of the module "Bachelor-Arbeit" in the department "Informatik" by

- Jeremy Stucki (jeremy.stucki@ost.ch)
- Vina Zahnd (vina.zahnd@ost.ch)

3. Introduction

This bachelor thesis is conducted for the external partner Sonova AG.

> *Sonova is a global leader in innovative hearing care solutions: from personal audio devices and wireless communication systems to audiological care services, hearing aids and cochlear implants. The Group was founded in 1947 and is headquartered in Stäfa, Switzerland.* [SON]

The Bluetooth Core Specification, since version 5.2, features LE Audio and Auracast to support audio transmission in different scenarios. They are advertised as

> *The next generation of Bluetooth audio - Building on 20 years of innovation, LE Audio enhances the performance of Bluetooth audio, adds support for hearing aids, and introduces Auracast (TM) broadcast audio, an innovative new Bluetooth use case with the potential to once again change the way we experience audio and connect with the world around us.* [BLU]

A typical Auracast setup is expected to consist of the following components:

- Audio sink, which receives and renders the audio signal.
- Broadcast transmitter, which is emitting an audio signal to be consumed by sinks.
- (Optional) Broadcast assistant, which helps with finding and selecting broadcasts, as well as configuring the sink to listen to it. This role is important as consumer devices which receive and play broadcast signals may be limited regarding their user interface and, therefore, might be limited in means of presenting available broadcast and selecting one.

For developing and especially testing audio devices that interact with such audio sources, it is necessary to have controllable counterparts in the test environment. The topic of this thesis focuses on providing the infrastructure to set-up test scenarios for audio devices receiving Bluetooth Auracast broadcasts.

4. Goals of the Project

In this bachelor thesis the students shall develop an extension of Sonova's test infrastructure to support testing of Auracast features in broadcast receivers. Specifically, test scenarios that consist of one - or if possible multiple – Auracast broadcasts, a broadcast assistant and broadcast receivers need to be supported. In the test environment the broadcast transmitter(s) and the broadcast assistant need to be controllable. The specific feature set is elaborated during the project and will be adapted according to the project progress in collaboration with the students.

The minimal feature set required to be offered by the developed component consists of:

- Provide one Auracast broadcast stream
- Scan for advertisement data to detect the audio sink device(s)
- Pair/bond with the audio sink device(s)
- Configure the audio sink as receiver through the broadcast assistant GATT service (BASS)

The device under test could be any Bluetooth LE audio/Auracast sink, like off-the-shelf headphones supporting it. For the broadcast transmitter and assistant Sonova suggests nRF5340 Audio Development Kits [NOR] and provides the corresponding hardware.

While the current environment for testing devices requires the implementation of a .NET-based library for controlling the external devices, integration into other environments should be possible as well. This should be considered when designing the interaction with the Bluetooth devices.

The progress of the team will be discussed in weekly meetings with the supervisor. Access to confidential or sensitive internal data of Sonova is regulated by a separate agreement with the students. The results of the thesis should be presented to an internal audience at Sonova at the end of the project.

5. Documentation

This project must be documented according to the guidelines of the "Informatik" department [SGI]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [AVT]. One printed copy of the documentation must be handed in (color, two-sided). Additional copies can be requested by the expert.

6. Important Dates

19.02.2024	**Start of the bachelor thesis.**
10.06.2024	**Hand-in of the abstract to the supervisor for checking. Information about accessing the corresponding web tool will be given by the department office.** **Hand-in (by email) of the A0 poster to the supervisor.**
14.06.2024 – 17:00 o'clock	**Final hand-in of the report**
TBD	**Presentation and Oral Exam**

7. Evaluation

A successful term project counts as 12 ECTS points. The estimated effort for 1 ECTS is 30 hours. (See also the module description [6]). The supervisor will be in charge of all the evaluation of the project.

Criterion	Weight
1. Organization, Execution	1/6
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization, and language of the whole documentation	1/6
3. Content	3/6
4. Final Presentation of the results and discussion	1/6

Furthermore, the general regulations for term projects of the department "Informatik" apply.

8. References

[SGI] https://ostch.sharepoint.com/:f:/r/teams/TS-StudienangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten
[AVT] https://avt.i.ost.ch (only available internally)
[SON] https://www.sonova.com/en/sonova-0 (01.02.2024)
[BLU] https://www.bluetooth.com/learn-about-bluetooth/feature-enhancements/le-audio/ (01.02.2024)
[NOR] https://www.nordicsemi.com/Products/Development-hardware/nRF5340-Audio-DK (01.02.2024)

Rapperswil, 16. February 2024

Thomas Corbat

18.4. Licenses

18.4.1. Google.Protobuf

```
Copyright 2008 Google Inc.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following disclaimer
in the documentation and/or other materials provided with the
distribution.
    * Neither the name of Google Inc. nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Code generated by the Protocol Buffer compiler is owned by the owner
of the input file used when generating it.  This code is not
standalone and requires a support library to be linked with it.  This
support library is itself covered by the above license.
```

Listing 19: Protobuf license