POLITECNICO DI TORINO

Master's Degree in Biomedical Engineering

Master's degree thesis

Wireless Bidirectional Real-Time Communication System for Intelligent Neural Interfaces

Supervisors

Ph.D. Paolo Motto Ros

M.Sc Andrea Dentis

Candidate

Stefano CONCADORO

DECEMBER 2024

Abstract

This thesis aims to create a wireless bidirectional real-time system enabling communication between an implantable device for neural signals acquisition and stimulation and an external mechatronic structure, such as an exoprosthesis or exoskeleton.

Two microcontroller boards handle wireless data transfer, one as a part of the implantable module and the other in the external control/signal processing module. Communication within each module is achieved via Serial Peripheral Interface (SPI), while the two modules communicate with each other over Bluetooth Low Energy (BLE). BLE technology, leveraging the Bluetooth 5 protocol, can ensure high data throughput. Custom firmware was developed for both modules, and the communication system was validated through dedicated testing.

Neural signal acquisition operates across five channels at 16-bit resolution and a 10 kHz transfer rate, requiring a data rate of at least 800 kbps. The neural stimulation system uses two channels at 16-bit resolution and a 1 kHz transfer rate, needing a minimum of 32 kbps. The optimal configuration achieved a throughput of at least 900 kbps from the implantable to the external module and 500 kbps concurrently in the opposite direction, meeting the specified requirements.

Contents

List of Figures

List of Tables

Glossary

ADC
Analog Digital Converter

BLE
Bluetooth Low Energy

AMTS
ATT MTU Throughput Service

CCCD
Client Characteristic Configuration Descriptor

CE
Connection Event

CI
Connection Interval

CLI
Command-Line Interface

CS
Chip Select

DAC
Digital Analog Converter

DLE
Data Length Extension

IDE
Integrated Development Environment

GAP
Generic Access Profile

GATT
Generic Attribute Profile

LBS
Led Button Service

MISO
Master Input, Serial Output

MOSI	Master Output, Serial Input Clock
SCLK	Serial Clock
MTU	Maximum Transmission Unit
SES	Segger Embedded Studio
SPI	Serial Peripheral Interface
SPIM	Serial Peripheral Interface Master
SPIS	Serial Peripheral Interface Slave
UUID	Universally Unique Identifier
WFE	Wait For Event

Chapter 1

Introduction

Recently, enhancements in prosthetic and human-machine interface fields have led to new perspectives on mobility and quality of life for patients with motor disease. Introducing exoprostheses and exoskeletons allows to overcome the limitations of traditional approaches, enabling recovery and functional integration with the human body. These devices, which combine avant-gard robotics, artificial intelligence, and biomechanics, represent cutting-edge rehabilitation and assistive care.

This project is part of NerveRepack, a co-funded European Union research to develop a new generation of bidirectional implantable electrodes that connect the human nervous system with assistive devices, such as exoskeletons and exoprostheses. The goal is to aid individuals with limb amputations or paralysis to regain motor and sensory functions. This approach aims to create prostheses controlled by the nervous system stimuli to enhance the support paradigm for people with disabilities. The project is expected to have a significant social, economic, medical, and technological impact, facilitating advancements in miniaturization, wireless communication and power supply, microsurgical tools, and the development of biocompatible materials. The project includes the Politecnico di Torino as one of its partners, who contribute expertise in wireless communication and wireless charging.

Integrating exoprostheses and exoskeletons with the human body is achieved through bidirectional communication with the nervous system. These devices capture signals from nerve bundles via plug electrodes implanted around the median and ulnar nerves. The signals collected by these electrodes are transmitted to an electronic module responsible for processing and directing them to the exoprosthesis, enabling movement control in response to the user's motor intentions. Simultaneously, the exoprosthesis is equipped with pressure sensors, allowing the user to have a sensory feedback. Consequently, the pressure signals collected by the exoprosthesis are transmitted wirelessly to the implanted module which triggers an appropriate response in the nervous system, giving the user a more natural sensation and an enhanced awareness of the device.

When the peripheral nervous system receives external stimuli through the prosthesis, it responds by generating action potentials: electrochemical impulses that propagate along nerve fibers as trains of pulses. Action potentials serve as a fundamental mechanism for neural communication and are characterized by a waveform with a typical duration of 1-2 milliseconds and an amplitude ranging from 70 to 100 mV. During propagation, the action potential signal assumes typical values that fluctuate between -70 and +30 mV. However, activity may extend beyond this range in cases of high-frequency impulses, such as those required for rapid and precise movements [16].

The work described in this thesis is organized into several sections to provide a comprehensive and detailed analysis of various aspects of the project. The thesis opens with this introductory section that outlines the study's objectives, situating the research within the broader context of human-machine interfaces and the current state of assistive technology. This is followed by a review of the state of the art, presenting previous studies and developments that have inspired this research. Subsequently, a description of the experimental setup and the software tools used for developing and managing device communication and control is provided. A dedicated section then examines the code developed and deployed on the boards, explaining each module's functionality and purpose. Finally, the results are analyzed, device performance is discussed, areas for potential improvement are identified, and suggestions for future developments are offered.

1.1 State of art

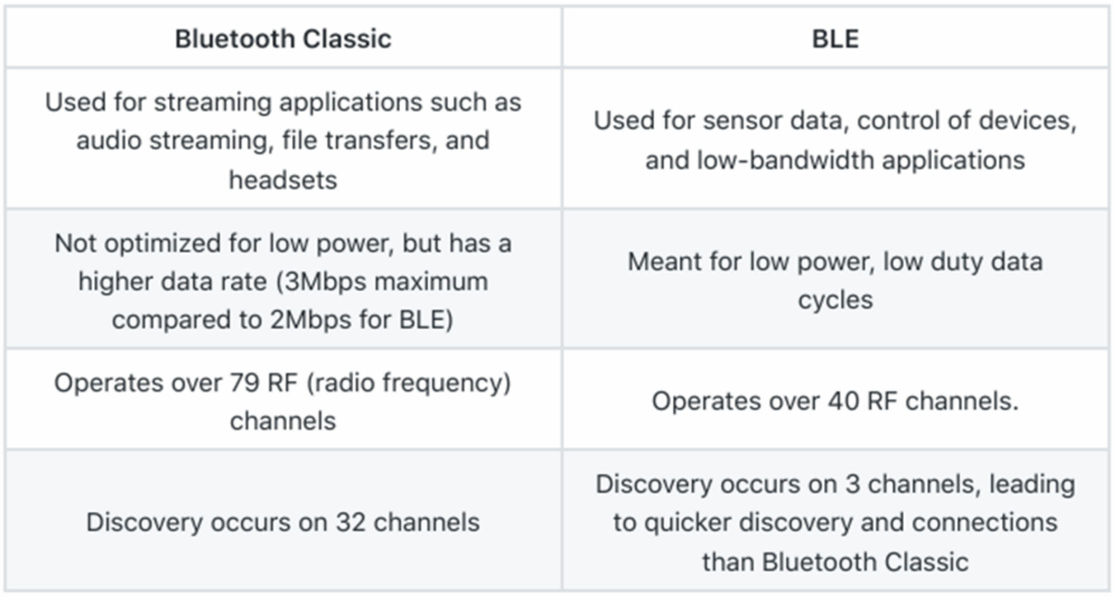
1.1.1 Basics of Bluetooth and BLE

Bluetooth technology, created and developed by Ericsson in 1994, has become omnipresent in modern wireless communication. Managed by the Bluetooth Special Interest Group (SIG), it has evolved into two primary categories: Bluetooth Classic and BLE.

Bluetooth Classic is optimized for continuous, high-bandwidth communication and is suitable for applications like audio streaming. On the other hand, BLE is designed for applications requiring low power consumption, such as wearable health monitors and Internet of Things (IoT) devices.

BLE was introduced in Bluetooth 4.0 to address the need for a protocol capable of supporting devices that require intermittent data transfers and long battery life. Like Bluetooth Classic, BLE operates in the 2.4 GHz Industrial, Scientific and Medical (ISM) band. However, it uses a more efficient frequency-hopping scheme, with 40 channels of 2 MHz width each, to reduce power consumption and improve resistance to interference. This efficiency is further enabled through various roles and parameters governing device interactions.

In Fig. 1.1 is shown a table that compares Bluetooth Classic with BLE.



Bluetooth Classic	BLE
Used for streaming applications such as audio streaming, file transfers, and headsets	Used for sensor data, control of devices, and low-bandwidth applications
Not optimized for low power, but has a higher data rate (3Mbps maximum compared to 2Mbps for BLE)	Meant for low power, low duty data cycles
Operates over 79 RF (radio frequency) channels	Operates over 40 RF channels.
Discovery occurs on 32 channels	Discovery occurs on 3 channels, leading to quicker discovery and connections than Bluetooth Classic

Figure 1.1: Bluetooth Classic vs BLE [1].

1.1.2 Device Roles in BLE: Central and Peripheral

BLE defines two primary roles for devices: Central and Peripheral. Central devices such as smartphones, tablets, or computers are typically more powerful. These devices have higher processing power and memory, making them well-suited to initiate and manage connections. A Central device actively scans for nearby Peripherals, establishes connections, and manages communication by sending read and write requests. Peripheral Devices, on the other hand, advertise their presence to nearby Central devices using low-power advertising to signal their availability, allowing them to operate efficiently even with limited energy resources.

1.1.3 Advertising and Connection Establishment

The process of communication between a Central and a Peripheral begins with advertising. Peripheral devices periodically broadcast advertising packets containing essential information, such as their device name, services offered, or a unique identifier. Based on the received data, Central devices scan for these advertisements and decide whether to connect.

Once a Central decides to connect, it starts a connection sending a request. Upon successful connection, the communication shifts from the advertising state to the active connected state, directed by parameters like Connection Event (CE) and Connection Interval (CI).

1.1.4 Connection Events and Parameters

CEs are the core of BLE communication. During these events, the Central and Peripheral devices exchange data. The CEs occur regularly, defined by the CI, which specifies the time between successive events. The Fig. 1.2 describes graphically the difference between CE and CI.

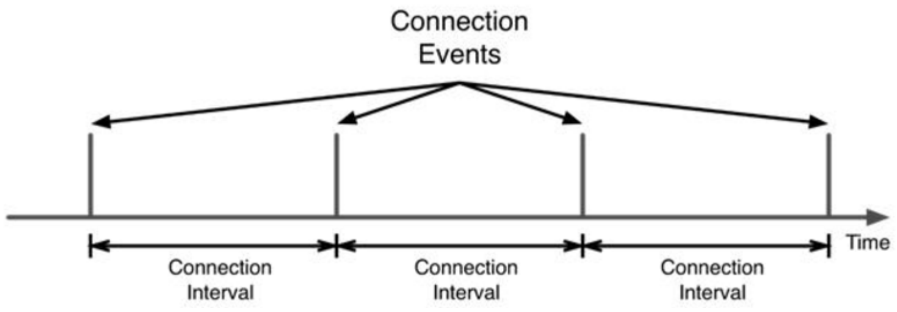


Figure 1.2: Graphical different between CE and CI [2].

The CI can be adjusted to balance power consumption and data throughput. A shorter interval allows for more frequent data exchanges but consumes more power, while a longer interval preserves energy but may introduce latency. Slave Latency is another parameter that helps manage power consumption. It defines the number of CEs a Peripheral can skip if it has no data to send, allowing it to enter a low-power state temporarily. This flexibility is crucial for energy efficiency, especially for battery-operated devices. Supervision Timeout is the maximum time a connection can remain inactive before termination. If no communication occurs within this timeout period, the connection is considered lost, leading both devices to reinitiate the connection process if needed.

1.1.5 BLE Architecture Overview

To better understand the BLE system, refer to the following diagram (Fig. 1.3), which illustrates the layered architecture:

The image depicts the BLE protocol stack, divided into three main components: Applications (or App Layer), Host and Controller.

Applications are at the top and represent software that utilizes BLE services, such as a mobile app for monitoring health metrics.

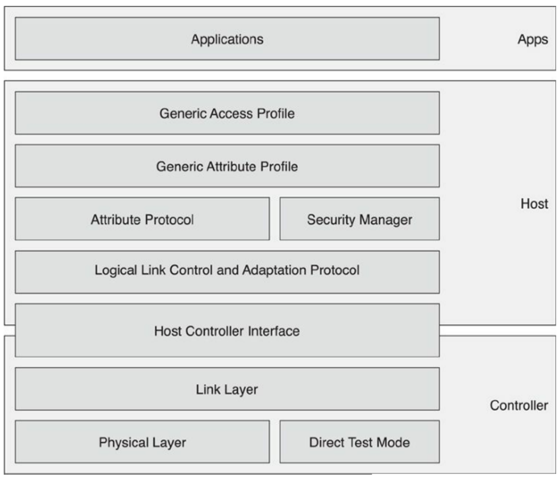


Figure 1.3: Bluetooth's architecture [3].

The host layer of the BLE architecture encompasses several critical protocols and profiles essential for managing data communication and ensuring security. The GAP (Generic Access Profile) and GATT (General Attribute Protocol) are responsible for device visibility and data structure organization. The Attribute Protocol (ATT) makes efficient data transfer possible, which defines the Server's data exposure to the Client, while the Security Manager (SM) ensures secure communication. The Logical Link Control and Adaptation Protocol (L2CAP) also handles the multiplexing of data streams, enabling data traffic management between connected devices.

The Controller layer manages the actual wireless transmission and offers several key components. The Host Controller Interface (HCI) facilitates communication between the Host and Controller, while the Link Layer (LL) and Physical Layer (PHY) manage data packet handling and radio transmission.

1.1.6 Data Exchange: GAP and GATT Profiles

As discussed in the BLE Architecture Overview section, BLE communication is governed by two critical profiles: GAP and GATT.

GAP handles the procedures for advertising, connecting, and broadcasting. It defines how BLE devices make themselves discoverable and how they establish connections. GAP outlines a device's roles (Central, Peripheral, Broadcaster or Observer) and manages the initial communication steps.

GATT defines how data is organized and exchanged once a connection is established. It uses a hierarchical structure of Services and Characteristics, where a Service is a collection of related data points, and a Characteristic represents an individual data item. For example, a heart rate monitor might have a Heart Rate

Service with a Characteristic for the actual heart rate measurement. GATT operates on a Client-Server model, with the Peripheral typically acting as the GATT Server that holds the data and the Central acting as the GATT Client that requests or modifies the data.

1.1.7 Optimizing for Low Power and Efficiency

BLE's low-power design is enabled by minimizing the time devices spend in active states and leveraging efficient connection management. Adaptive Frequency Hopping ensures robust communication in the congested 2.4 GHz band by rapidly switching between channels to avoid interference. The Fig. 1.4 illustrates the division of the physical channel within the 2.4 GHz ISM band utilized by BLE.

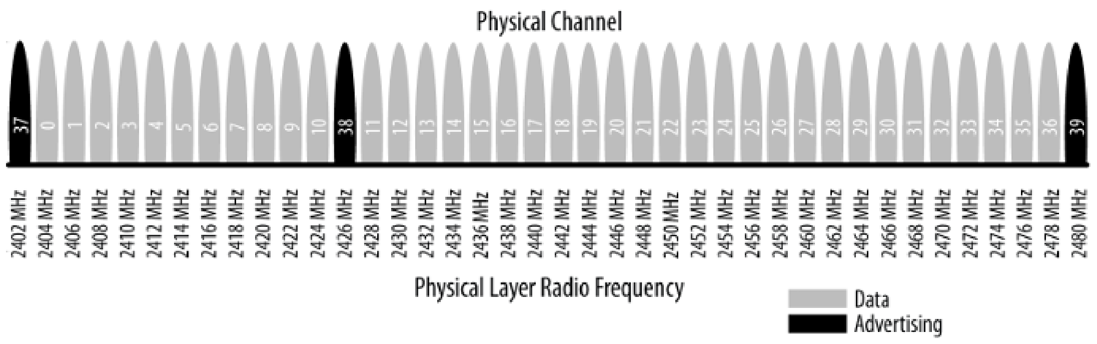


Figure 1.4: Adaptive Frequency Hopping [3].

This band is segmented into 40 channels, each 2 MHz wide. Three channels (37, 38, and 39), highlighted in black, are designated for advertising. These channels are used by devices to broadcast advertising packets and announce their presence to Central devices. The remaining 37 channels, shown in gray, are allocated for data transmission once a connection has been established. With dedicated advertising channels, this channel allocation enhances the efficiency of scanning and connection processes, minimizing interference in crowded environments. Furthermore, BLE employs Adaptive Frequency Hopping, which enables devices to switch between these channels to ensure reliable communication even in the presence of other devices using the same frequency band.

1.1.8 Bluetooth 4.2

Bluetooth 4.2, released in December 2014, is a significant milestone in the evolution of Bluetooth technology. The SIG introduced Data Length Extension (DLE) in the Bluetooth 4.2 Core Specification, allowing maximum data rate. For Bluetooth 4.2 and earlier, the rate is fixed at 1 Mbps [1], making communication between devices more efficient and suitable for applications requiring faster data exchange. This advancement reduced the transmission time and made Bluetooth more competitive in environments where performance is crucial.

DLE is a feature added to the Link Layer that allows the Data Channel Protocol Data Unit (PDU) Payload field to be increased from the default 27 bytes up to 251 bytes [4]. This ability reduces the overhead associated with data transmission, enhancing the overall throughput and efficiency of BLE communications. By enabling larger payloads to be sent in a single packet, DLE minimizes fragmentation and reduces the time required for data transfer, which is particularly beneficial for applications involving large datasets.

Below is shown the difference on Data Channel Payload without DLE (Fig. 1.5) and with DLE (Fig. 1.6).

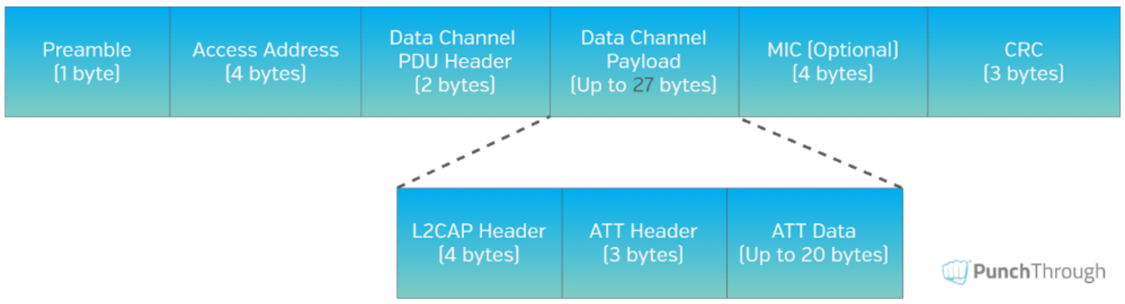


Figure 1.5: Link Layer Packet Structure for an L2CAP Start packet of max size without DLE [4].

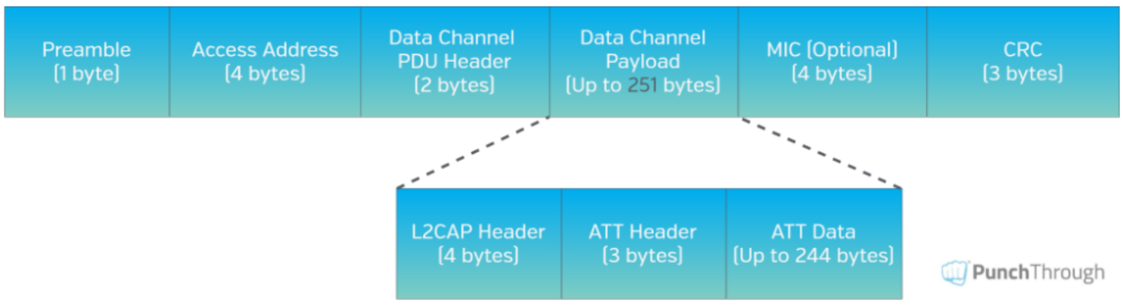


Figure 1.6: Link Layer Packet Structure for an L2CAP Start packet of max size without DLE [4].

The influence of DLE on better managing CEs is understood well in the images below, first without DLE (Fig. 1.7) and with DLE (Fig. 1.8):

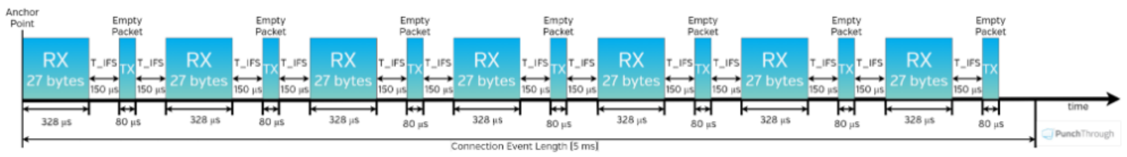


Figure 1.7: A single CE (from the slave's perspective) in a connection without DLE, in which the master is transmitting as much data as possible within the effective CE Length [4].

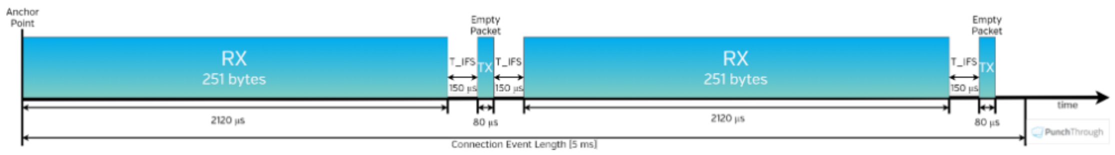


Figure 1.8: A single CE (from the slave's perspective) in a DLE-enabled connection in which the master is transmitting as much data as possible within the effective CE Length [4].

1.1.9 Bluetooth 5

Bluetooth 5, introduced in December 2016, brought transformative enhancements that significantly expanded the potential of BLE technology. The most notable advancements was doubling the maximum data transfer rate to 2 Mbps. This improvement enabled faster and more reliable data exchange, which is critical for applications requiring real-time communication, such as those in advanced medical and assistive devices. While the theoretical maximum speed of 2 Mbps is impressive, practical implementations typically achieve effective data rates closer to 1.4 Mbps.[1].

Moreover, Bluetooth 5 optimizes energy consumption, a critical factor for battery-operated devices like wearables and implantables, ensuring that devices with limited power resources can operate without compromising performance. Bluetooth 5 has become a revolutionary wireless technology that pushes the boundaries of what is possible for energy-efficient, high-performance communication systems in the biomedical field and beyond.

1.1.10 Basics of SPI

Synchronous serial communication protocol SPI is widely used in embedded systems for short-distance communication. Full-duplex mode allows simultaneous data transmission and reception between a master device and one or more slave devices [15].

The SPI buses consist of four primary signals

- Serial Clock (SCLK): Generated by the Master to synchronize data transmission.
- Master Out, Slave In (MOSI): Carries data from the Master to the Slave.
- Master In, Slave Out (MISO): Carries data from the slave to the master.
- Chip Select(CS): Used by the master to select a specific slave device for communication.

Compared to other communication protocols, SPI offers several advantages

- high Speed: SPI can operate at higher data rates than protocols like I^2C, making it suitable for applications requiring fast data transfer;
- full-duplex communication: SPI allows simultaneous data transmission in both directions, enhancing communication efficiency, unlike I^2C which is half-duplex;
- simplicity and Low Overhead: Minimal communication overhead is achieved by SPI's straightforward protocol structure, which leads to faster data exchange.

However, SPI also has some limitations

- limited Distance: The purpose of SPI is to provide short-distance communication [15], usually within a single device or between nearby devices;
- no Acknowledgment Mechanism: SPI's absence of an acknowledgment feature makes error detection more difficult, unlike I2C;
- multiple Slave Management: managing multiple Slave demands additional chip select lines, which can complicate hardware design.

Despite these limitations, SPI's high speed and full-duplex capabilities make it a suitable choice for the data transmission requirements of this project.

1.1.11 Relevant Studies

The history of prosthetics dates back to antiquity. One of the oldest known examples is an artificial wooden and leather toe, dated between 950 and 710 BCE, found on an Egyptian mummy near Luxor.

Regarding neuroprosthetics, the first cochlear implant was developed in 1957, marking the beginning of electronic devices designed to restore compromised sensory functions.

Recent studies have demonstrated that integrating bionic sensors into prosthetics can replicate sensations such as touch, warmth, and cold, offering users a more natural perception of their surroundings. For instance, researchers at the École Polytechnique Fédérale de Lausanne (EPFL) developed a technology called MiniTouch, which enables amputees to sense temperature changes in their phantom hand by transferring thermal information from the prosthetic's fingers to the residual limb [5].

In Fig. 1.9, it is possible to appreciate the thermal sensor on the prosthetic finger and its view using a thermal camera.

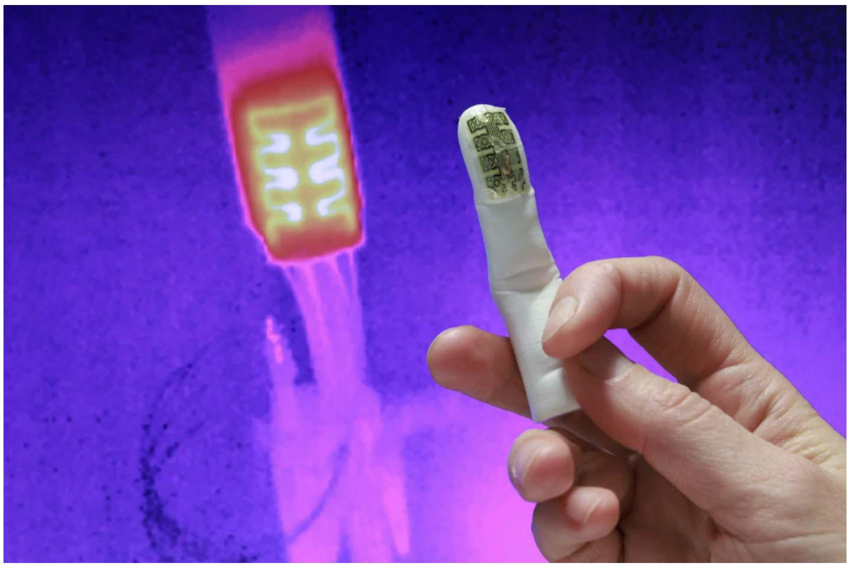


Figure 1.9: Vision of the thermal sensor using a thermal camera [5].

Modern prosthetic devices increasingly incorporate bidirectional wireless communication systems based on Bluetooth Low Energy (BLE) technology to enhance user-device interaction (as is shown in Fig. 1.10, it is possible to appreciate the thermal sensor on the prosthetic finger and its view using a thermal camera. BLE offers low power consumption and robust data transmission, making it suitable for real-time control and feedback in prosthetic applications. For instance, a study published in Frontiers in Neuroscience in 2018 discusses the development of totally implantable bidirectional neural prostheses that utilize wireless data transmission for both neurostimulation and neural sensing, highlighting the role of BLE in facilitating these functions [6].

A relevant article in neuroprosthetics is Raspopovic et al.'s work [7]. This study highlights the importance of neural function restorage in prosthetic development, especially through BLE for efficient real-time bidirectional communication. BLE technology was used by the researchers to transmit tactile and proprioceptive feedback to users, allowing for natural and responsive interactions in the surroundings.

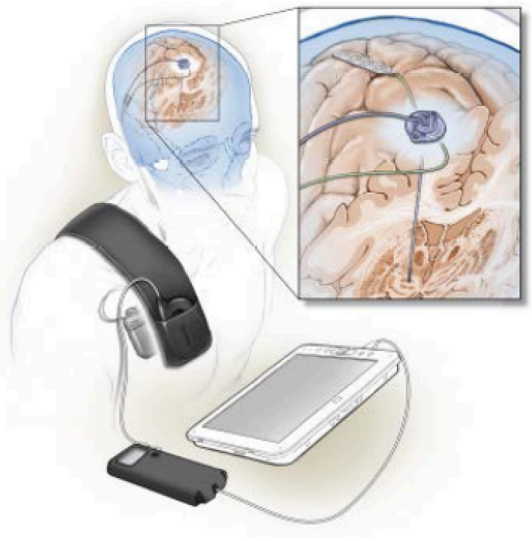


Figure 1.10: The figure shows a totally implanted bidirectional neural interface designed for chronic multisite recording in humans, as well as therapeutic neurostimulation [6].

This approach shows that advanced prosthetic hands can function more effectively with BLE's continuous, low-power wireless communication between neural interfaces and prosthetic components. This is crucial for creating a more life-like user experience and promoting precise control in real-world tasks.

In Fig. 1.11(A) is shown the general setup of the study, then in In Fig. 1.11(B) surgical insertion of the electrode in the median nerve and also, in Fig. 1.11(C) the ulnar nerve with the implanted electodes

Kim et al., 2020 [8], published an article that enabled advancements in the research on real-time wireless monitoring of implantable devices (Fig. 1.12). The article highlights the fact that BLE is designed for minimal power usage, consuming between 0.01 and 0.5 W, which is especially advantageous for implantable devices, as it reduces the need for bulky power sources and enables prolonged operation without requiring battery replacement, thereby avoiding health risks and the need for surgical intervention. Additionally, the article emphasizes the compatibility of BLE with most consumer devices, including smartphones and tablets, which facilitates the integration of implantable devices into patients' daily lives and allows them to monitor their physiological parameters in real-time without relying on dedicated medical equipment. Finally, the ability of BLE to efficient data collection, encoding, and transmission to external devices is cited, which is essential for continuous health monitoring, as it allows data to be sent in real-time to healthcare providers or monitoring systems, enhancing diagnosis and disease management.

For the realization of wireless implantable devices, it is necessary to consider

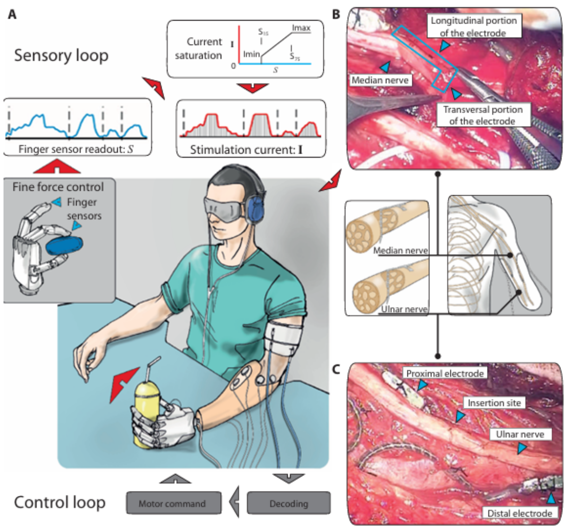


Figure 1.11: (A) The current was delivered as a function of the prosthetic hand sensor readouts. (B) Photograph of the surgical insertion of a TIME electrode in the median nerve of the participant. (C) Depiction of the subject's ulnar nerve with the two implanted electrodes [7].

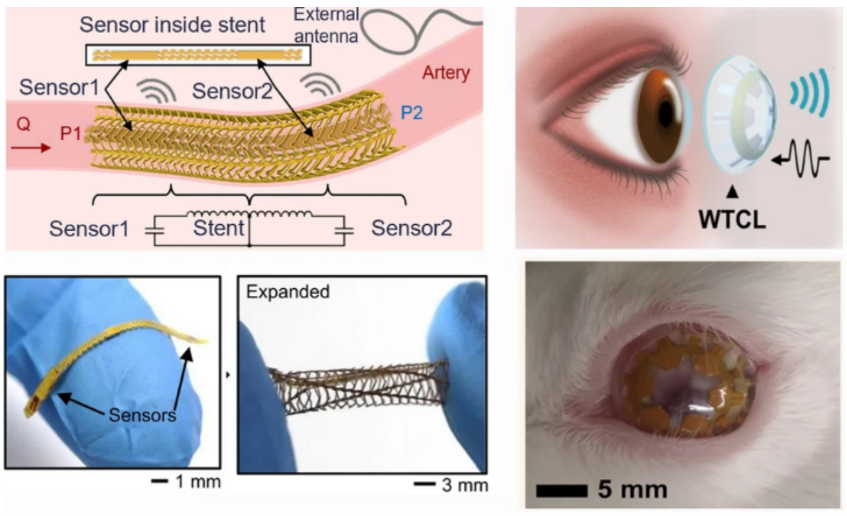


Figure 1.12: At the left, implantable pressure sensors designed to fit into an inductive stent. At the right, Sensor for monitoring IntraOcular Pressure (IOP) integrated into a contact lens [8].

both efficient data transfer and safe power delivery through biological tissues. The attenuation of electromagnetic signals occurs as they pass through tissue, which absorbs part of the signal energy. The Specific Absorption Rate (SAR) quantifies this phenomenon by measuring the energy absorbed by the body from electromagnetic waves depending on the tissue type and signal frequency. Compliance with SAR safety limits is essential to prevent excessive heat generation that could damage surrounding tissue. Different organs have different sensitivities to heat; for example, the eye and testicles are more susceptible to damage than other, more resistant tissues. For this purpose, Kim et al. dedicate a section about the SAR limit (Fig. 1.13). In fact, one of the primary constraints in determining the implantation depth of wireless devices is the SAR, as deeper implants are subject to more excellent energy absorption by the surrounding tissue, thereby increasing the SAR and reducing the energy reaching the implantable sensor. Devices like these were designed to maximize power delivery while remaining within SAR safety limits to address these challenges. Kim et al. cite studies that include comprehensive SAR calculations and measurements as a reference for research to ensure safety compliance.

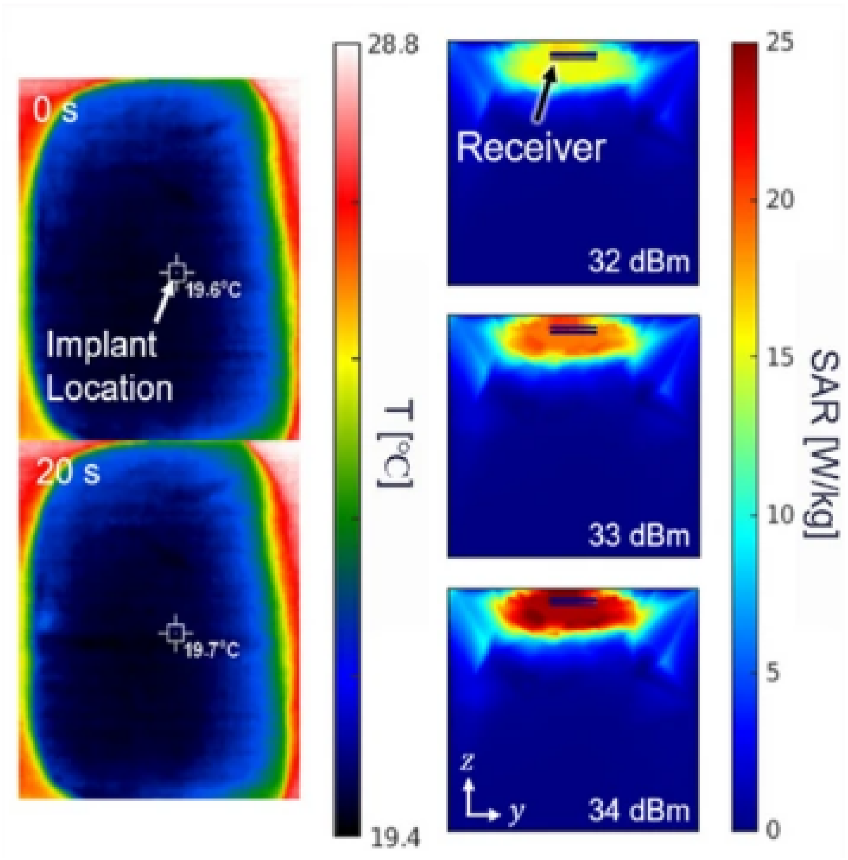


Figure 1.13: Safety test in a piece of pork and SAR simulation of implantable device [8].

One study particularly relevant to the state of the art in bidirectional communication for prosthetic applications is the article titled "SenseBack - An Implantable System for Bidirectional Neural Interfacing" [9]. This research presents the SenseBack system, which enables bidirectional neural interfacing and is specifically designed for chronic peripheral electrophysiology experiments. The system employs Bluetooth 5 (as it is possible to see from the Bluetooth microcontroller present in the scruff in Fig. 1.14) for data communication, achieving real-time wireless data transfer with low latency. The setup includes a miniaturized, implantable module capable of recording neural signals and performing high-voltage stimulation on any of its 32 channels. Also, this pioneering system demonstrates the feasibility of using BLE for continuous, efficient, and low-power data exchange between implanted devices and external modules, providing a promising approach for future developments in prosthetics and neural interfaces.

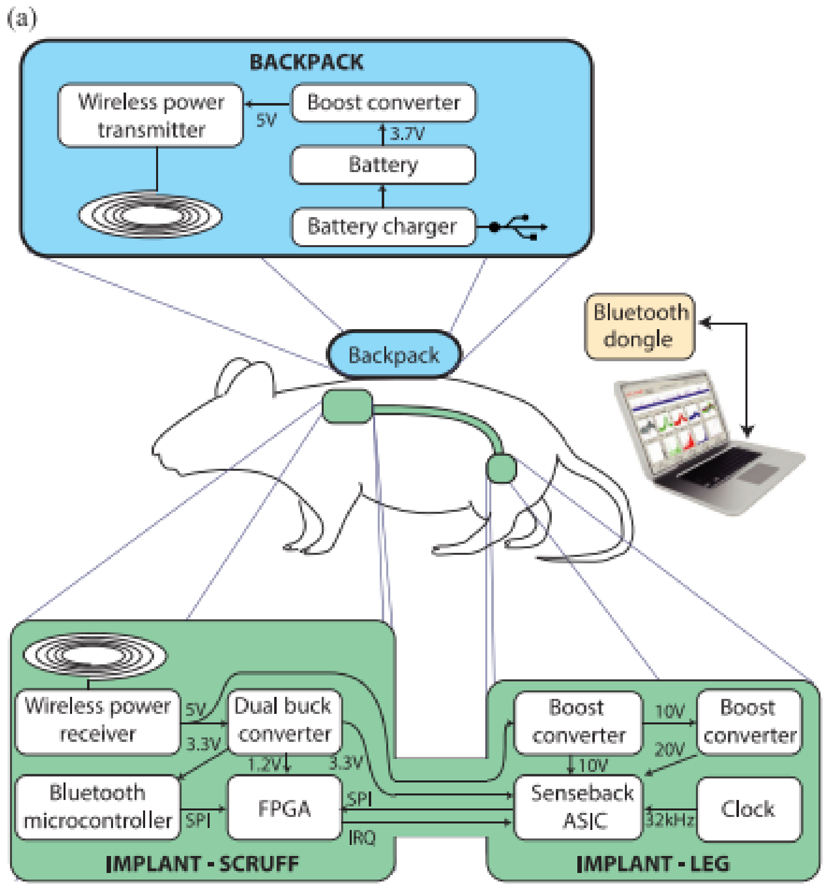


Figure 1.14: The block diagram of the overall system [9].

The restoration of tactile and proprioceptive sensation is essential due to the importance of effective bi-directional communication between the nervous system and prostheses, as highlighted by these advances. The development of low-power wireless communication systems has been facilitated by adopting technologies like BLE, which has enhanced the interaction between the user and the prosthesis.

1.2 Aim of the study

The primary objective of this thesis is to develop a code that supports a real-time bidirectional wireless system using BLE technology between two microcontrollers. One of these microcontrollers simulates the implantable device. This device also allows the signal acquisition from nerves (Fig. 1.15).

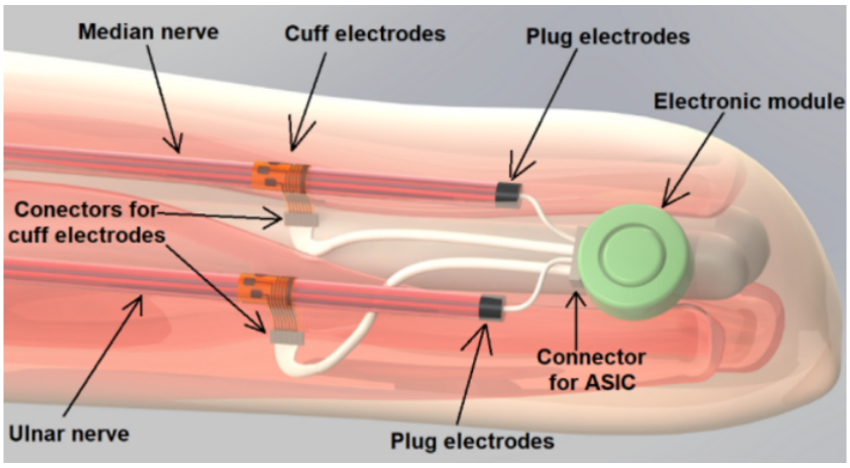


Figure 1.15: Illustration of an implantable neural interface system, showing cuff electrodes around the median and ulnar nerves and the electronic module [10].

On the other hand, the external module represents the control unit, which takes the signals from pressure sensors to transmit via BLE to the internal module. At the same time, the external module receives the movement information for the exoprosthesis or exoskeleton from the internal module (Figure 1.16).

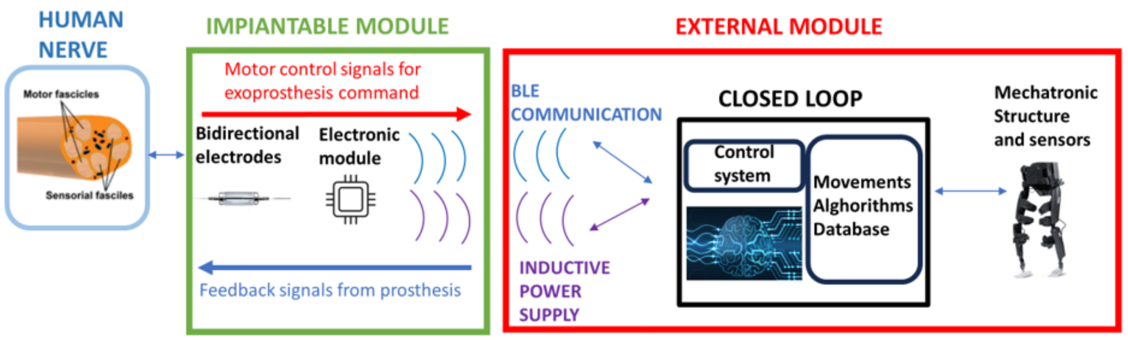


Figure 1.16: Diagram illustrating the bidirectional communication system between the implantable and external modules for controlling a mechatronic structure.

The importance of real-time capability in this project lies in replicating the seamless and immediate processing of information that we naturally experience through our nervous system, enabling rapid responses and decision-making. Similarly, this project aims to replicate the immediacy and fluidity of communication inherent in the neural interface. For this purpose, some specifications projects to respect are given.

1.3 Requirements

Because of the importance of the real-time necessity explained in the Aim of the study, some specifications are required.

The neural signal acquisition exploits an Analog Digital converter (ADC) at 13-bit, which converts the analog signals detected from the nerves into digital form using 13-bit resolution. It is also known that the system captures neural signals at a rate of 10.000 samples per second and also that this happens through five channels at 16 bit each, correspondent to the five plugs of the plug electrodes on the medial and ulnar nerves (three of these five plug electrodes on the ulnar nerves and the others two on the ulnar one), as shown in Fig. 1.17.

The neural stimulation data occurs thanks to cuff electrodes positioned as the plug electrodes, as possible to see in Fig. 1.17 on the ulnar and medial nerves. The stimulation occurs thanks to a 9-bit Digital Analog Converter (DAC) to convert the digital neural stimulation data into an analog signal to stimulate the nerves. The stimulation signals are sampled at 10kHz, and two separate channels are used to send the signal. The stimulation signal is active for 10% of the time and inactive for the remaining 90% in order to obtain a duty cycle of 10%.

So, that, it is possible to obtain the requirements as Eq.1 and Eq.2:

- for neural signal acquisition: at least 800 kbps, since it is used at a 10 kHz sampling rate, with five channels at 16 bits.

$$5 \text{ channels} \times 16 \text{ bits} \times 10 \text{ kHz} = 800 \text{ kbps} \tag{1.1}$$

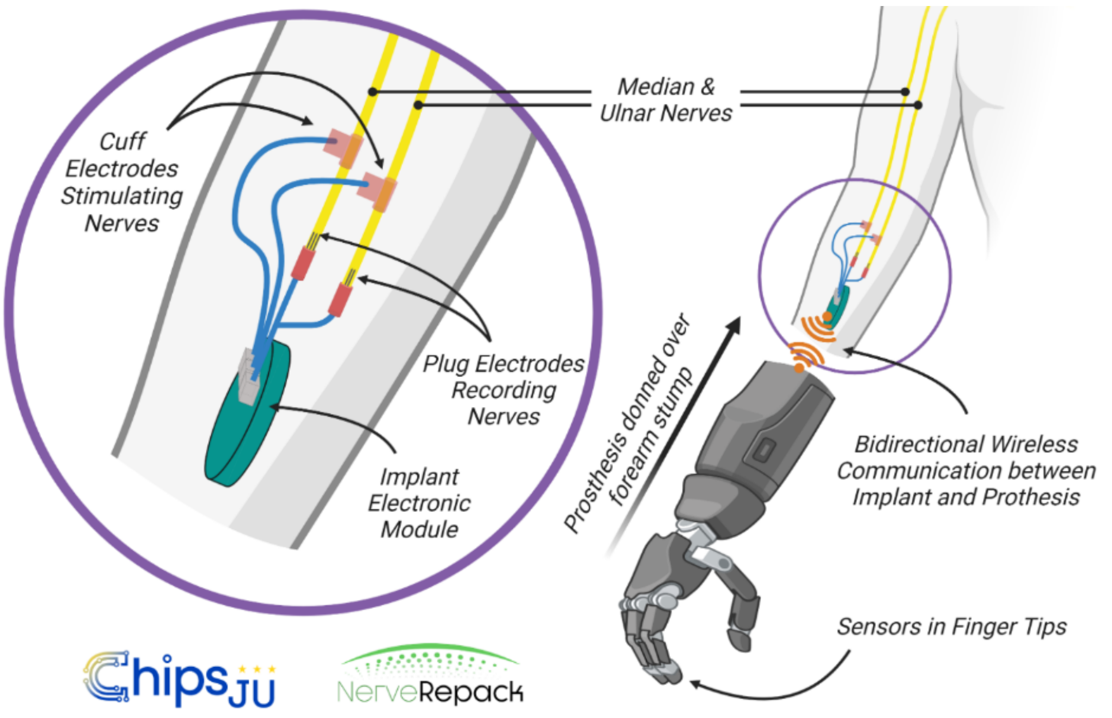

Cuff Electrodes Stimulating Nerves
Median & Ulnar Nerves
Plug Electrodes Recording Nerves
Implant Electronic Module
Prosthesis donned over forearm stump
Bidirectional Wireless Communication between Implant and Prothesis
Sensors in Finger Tips
ChipsJU
NerveRepack

Figure 1.17: In this image is possible to distinguish the five plugs of the two plugs electrodes and the two cuff electrodes [11].

- for the neural stimulation data: at least 32 kbps data rate, using two channels each at 9 bit at 10 kHz sampling rate and 10% duty cicle.

$$2 \text{ channels} \times 16 \text{ bits} \times 10 \text{ kHz} \times 0.1 = 32 \text{ kbps} \tag{1.2}$$

Real-time performance only sometimes guarantees data integrity, as communication systems may prioritize transmission speed over data accuracy. This thesis also addresses the measures implemented to overcome this limitation, ensuring high performance while maintaining data integrity.

Chapter 2

Methods

In this section, it is possible to understand the role of each board and how the softwares used interact with them.

2.1 Setup

The nRF52832 [17] and nRF52840 [18] are BLE System-on-Chips (SoCs) developed by Nordic Semiconductor, widely utilized in wireless communication applications. The nRF52832 is a general-purpose BLE SoC featuring a 32-bit ARM Cortex-M4 CPU with a floating-point unit, operating at 64 MHz. It offers 512 kB of flash memory and 64 kB of RAM, supporting various peripherals such as NFC, SPI, I2C, UART, and PWM. Known for its low power consumption, the nRF52832 is ideal for battery-operated devices. Its advantages include low power consumption suitable for energy-sensitive applications, comprehensive peripheral support enabling different functionalities, and being a cost-effective solution for standard BLE applications. However, it has some limitations, such as limited memory capacity that may constrain complex applications and lacks certain advanced features present in newer models.

In contrast, the nRF52840 is a more sophisticated BLE SoC that includes a 32-bit ARM Cortex-M4 processor and a floating-point unit, and it operates at 64 MHz. With 1 MB of flash memory and 256 kB of RAM, it supports multiple protocols such as BLE, Thread, Zigbee, and ANT. Key benefits of the nRF52840 include increased memory capacity to support more complex applications, multi-protocol support for wireless communication versatility, integrated USB controller for direct USB connectivity and enhanced security features for robust data protection. Compared to the nRF52832, the power consumption is higher and the cost is higher because of its advanced features.

The nRF52 DK and nRF52840 DKs were extensively used in this project [19, 20]. By providing access to all GPIO pins, LEDs, and buttons, these kits make hardware interfacing and debugging easier. Programming and debugging of the SoCs is perfect with their onboard SEGGER J-Link debugger. Flexible testing scenarios can be achieved using various power sources, such as USB, external power, and batteries, with the DKs. The advantages of using these DKs are simplified developing processes, full access to SoC features, integrated debugging tools and support for different power configurations. By using these development kits, the project takes advantage of an efficient development workflow that allows thorough testing and validation of the wireless communication system.

Below a graphical scheme of the theoretical (Fig. 2.1) and real (Fig. 2.2) setup:

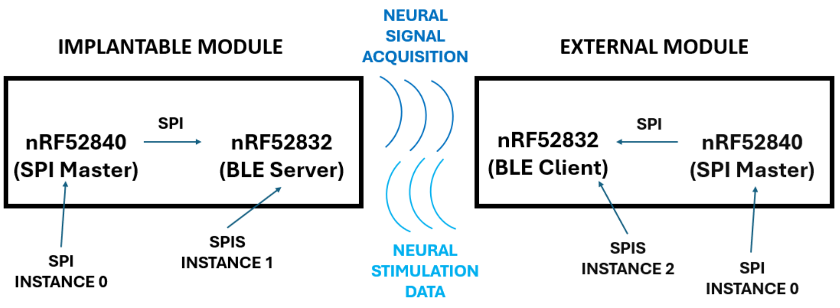


Figure 2.1: Graphic scheme of the setup.

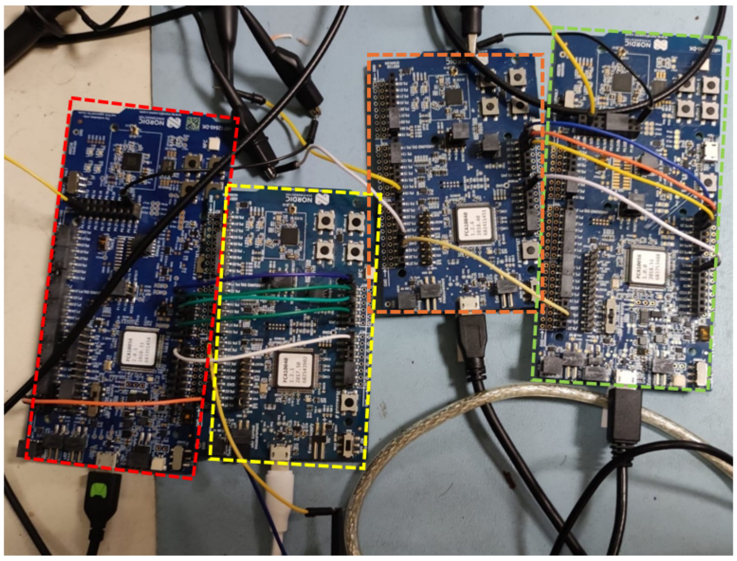


Figure 2.2: The real setup. In red the SPI Master External, in yellow the BLE External, in orange the BLE Implantable and in green the SPI Master External.

As we can see in Fig. 2.2, the boards are two by two connected via jumpers. Those connections define the pins for SPI communication: MISO, MOSI, SCLK and CS. This way, are locable two Master SPI and two Slave SPI. The two SPI

Masters are the nRF52840 boards and the two SPI Slaves are the nRF52832 boards. The Slave SPI also acts as a BLE Server (BLE Implantable) and BLE Client (BLE External). In this way, the nRF52832 cards communicate with each other via BLE the data packets received via SPI from the respective nRF52840 boards.

2.2 Softwares and Hardwares

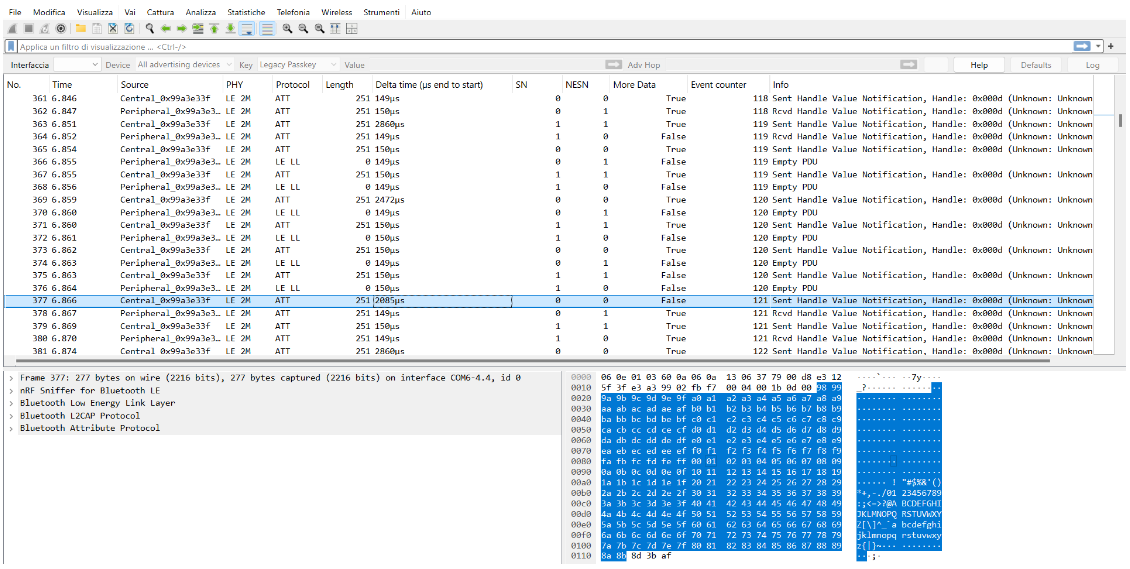
This project used different software and additional hardware to facilitate development and testing.

The combination of these tools provided a robust environment for developing, testing, and validating the wireless bidirectional real-time communication system implemented in this project.

2.2.1 Software

SEGGER Embedded Studio (SES) [21] is an Integrated Development Environment (IDE) employed for writing, compiling, and debugging code on the Nordic Semiconductor boards used in the project. It offers comprehensive support for ARM Cortex devices, making it suitable for the nRF52832 and nRF52840 microcontrollers.
For this project the 5.42a version of SES is used.

Another software used in the project was Wireshark [12], which is a network protocol analyzer used in conjunction with a BLE sniffer to monitor and analyze BLE communication. This setup allowed for the verification of data packet integrity during wireless transmission, catching the BLE channel where Implantable and External communicate. In Fig. 2.3 some packets exchanged between the Implant and External are shown.
In this project the stable release 4.4.1 was used.



```
No.  Time   Source                 PHY    Protocol Length Delta time (µs end to start) SN NESN More Data Event counter Info
 361 6.846  Central_0x99a3e33f     LE 2M  ATT      251 149µs   0  0  True   118 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 362 6.847  Peripheral_0x99a3e3…   LE 2M  ATT      251 150µs   0  1  True   118 Rcvd Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 363 6.851  Central_0x99a3e33f     LE 2M  ATT      251 2860µs  1  1  True   119 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 364 6.852  Peripheral_0x99a3e3…   LE 2M  ATT      251 149µs   1  0  False  119 Rcvd Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 365 6.854  Central_0x99a3e33f     LE 2M  ATT      251 150µs   0  0  True   119 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 366 6.855  Peripheral_0x99a3e3…   LE 2M  LE LL      0 149µs   0  1  False  119 Empty PDU
 367 6.855  Central_0x99a3e33f     LE 2M  ATT      251 150µs   1  1  True   119 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 368 6.856  Peripheral_0x99a3e3…   LE 2M  LE LL      0 149µs   1  0  False  119 Empty PDU
 369 6.859  Central_0x99a3e33f     LE 2M  ATT      251 2472µs  0  0  True   120 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 370 6.860  Peripheral_0x99a3e3…   LE 2M  LE LL      0 149µs   0  1  False  120 Empty PDU
 371 6.860  Central_0x99a3e33f     LE 2M  ATT      251 150µs   1  1  True   120 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 372 6.861  Peripheral_0x99a3e3…   LE 2M  LE LL      0 150µs   1  0  False  120 Empty PDU
 373 6.862  Central_0x99a3e33f     LE 2M  ATT      251 150µs   0  0  True   120 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 374 6.863  Peripheral_0x99a3e3…   LE 2M  LE LL      0 149µs   0  1  False  120 Empty PDU
 375 6.863  Central_0x99a3e33f     LE 2M  ATT      251 150µs   1  1  False  120 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 376 6.864  Peripheral_0x99a3e3…   LE 2M  LE LL      0 150µs   1  0  False  120 Empty PDU
 377 6.866  Central_0x99a3e33f     LE 2M  ATT      251 2085µs  0  0  False  121 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 378 6.867  Peripheral_0x99a3e3…   LE 2M  ATT      251 149µs   0  1  True   121 Rcvd Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 379 6.869  Central_0x99a3e33f     LE 2M  ATT      251 150µs   1  1  True   121 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 380 6.870  Peripheral_0x99a3e3…   LE 2M  ATT      251 149µs   1  0  True   121 Rcvd Handle Value Notification, Handle: 0x000d (Unknown: Unknown
 381 6.874  Central_0x99a3e33f     LE 2M  ATT      251 2860µs  0  0  True   122 Sent Handle Value Notification, Handle: 0x000d (Unknown: Unknown
```

```
> Frame 377: 277 bytes on wire (2216 bits), 277 bytes captured (2216 bits) on interface COM6-4.4, id 0
> nRF Sniffer for Bluetooth LE
> Bluetooth Low Energy Link Layer
> Bluetooth L2CAP Protocol
> Bluetooth Attribute Protocol
```

```
0000  06 0e 01 03 60 0a 06 0a  13 06 37 79 00 d8 e3 12   ····`··· ··7y····
0010  5f 3f e3 a3 99 02 fb f7  00 04 00 1b 0d 00 98 99   _?······ ········
0020  9a 9b 9c 9d 9e 9f a0 a1  a2 a3 a4 a5 a6 a7 a8 a9   ········ ········
0030  aa ab ac ad ae af b0 b1  b2 b3 b4 b5 b6 b7 b8 b9   ········ ········
0040  ba bb bc bd be bf c0 c1  c2 c3 c4 c5 c6 c7 c8 c9   ········ ········
0050  ca cb cc cd ce cf d0 d1  d2 d3 d4 d5 d6 d7 d8 d9   ········ ········
0060  da db dc dd de df e0 e1  e2 e3 e4 e5 e6 e7 e8 e9   ········ ········
0070  ea eb ec ed ee ef f0 f1  f2 f3 f4 f5 f6 f7 f8 f9   ········ ········
0080  fa fb fc fd fe ff 00 01  02 03 04 05 06 07 08 09   ········ ········
0090  0a 0b 0c 0d 0e 0f 10 11  12 13 14 15 16 17 18 19   ········ ········
00a0  1a 1b 1c 1d 1e 1f 20 21  22 23 24 25 26 27 28 29   ······ ! "#$%&'()
00b0  2a 2b 2c 2d 2e 2f 30 31  32 33 34 35 36 37 38 39   *+,-./01 23456789
00c0  3a 3b 3c 3d 3e 3f 40 41  42 43 44 45 46 47 48 49   :;<=>?@A BCDEFGHI
00d0  4a 4b 4c 4d 4e 4f 50 51  52 53 54 55 56 57 58 59   JKLMNOPQ RSTUVWXY
00e0  5a 5b 5c 5d 5e 5f 60 61  62 63 64 65 66 67 68 69   Z[\]^_`a bcdefghi
00f0  6a 6b 6c 6d 6e 6f 70 71  72 73 74 75 76 77 78 79   jklmnopq rstuvwxy
0100  7a 7b 7c 7d 7e 7f 80 81  82 83 84 85 86 87 88 89   z{|}~··· ········
0110  8a 8b 8d 3b af                                     ···;·
```

Figure 2.3: A capture of some packets on Wireshark [12].

The BLE sniffer is an nRF52840 Dongle[13] and is shown in Fig. 2.4.

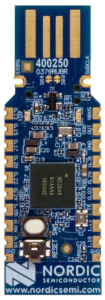
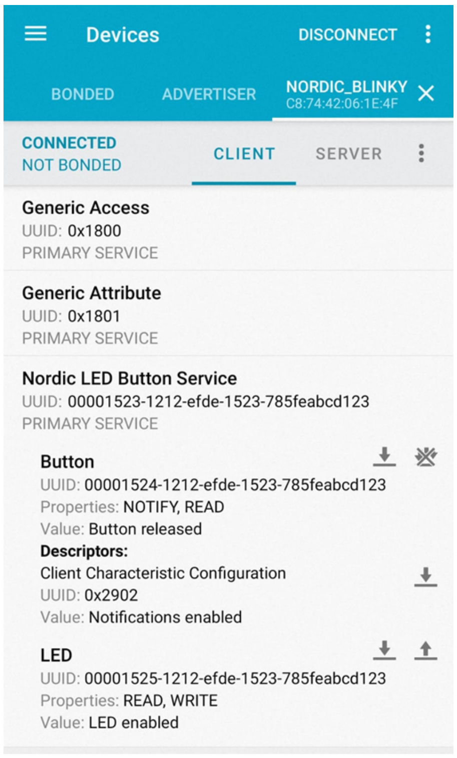


Figure 2.4: nRF52840 Dongle [13].

nRF Connect for Desktop [22] is a cross-platform tool framework designed to assist in the development of applications for nRF devices. It includes various applications for testing, monitoring, measuring, optimizing, and programming, which were particularly useful during the initial stages of the project.

For this program, it is advised to use the latest version.

Another tool used in preliminary phases was nRF Connect for mobile[14]. It was very useful to gain confidance individually with each board, using BLE communication to connect the smartphone with the boards. In Fig. 2.5 is shown a screenshoot which depicts the use of the LED Button Service (LBS).



Devices DISCONNECT

BONDED ADVERTISER NORDIC_BLINKY C8:74:42:06:1E:4F

CONNECTED
NOT BONDED CLIENT SERVER

Generic Access
UUID: 0x1800
PRIMARY SERVICE

Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Nordic LED Button Service
UUID: 00001523-1212-efde-1523-785feabcd123
PRIMARY SERVICE

Button
UUID: 00001524-1212-efde-1523-785feabcd123
Properties: NOTIFY, READ
Value: Button released
Descriptors:
Client Characteristic Configuration
UUID: 0x2902
Value: Notifications enabled

LED
UUID: 00001525-1212-efde-1523-785feabcd123
Properties: READ, WRITE
Value: LED enabled

Figure 2.5: Screenshoot using nRF Connect for mobile [14].

A key piece of software was PuTTY. PuTTY facilitated firmware debugging through the integration of the Command-Line Interface (CLI), which allows commands to be sent and responses to be received from the connected device. In this case, the connection between PuTTY and the boards was serial, via COM ports.

2.2.2 Hardware

The signals from the boards are taken through the digital oscilloscope Rigol MSO5104 and the probes in 1x.

Figure 2.6: Digital oscilloscope Rigol MSO5104.

SPIDriver [15] provides a perfectly understanding of SPI communication. This tool was used in the first steps of the project to simulate and verify the behaviour of digital circuits before hardware implementation.

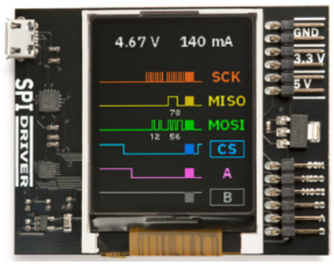


Figure 2.7: SPIDriver [15] .

The SPIDriver by Excamera Labs was utilized. This tool enables the control of SPI devices through a graphical interface on a PC

Chapter 3

Development and Testing of Codes Implementations

In this section, will be shown a step-by-step explaination of the implemented code to enable a complete understanding of the development of the system. The carried-out tests will be shown to ensure that the system works and meets the requirements.

The codes running on the four microcontrollers are divided on basis of the role of the board:

- the two nRF52840, as SPI Masters, have two similar codes based on the example code "spi" of the Nordic;
- the two nRF52832 share a code which contains *main.c*, *amts.c* and *amtc.c*, as well as necessary functions. The BLE microcontroller for the Implantable module uses *amts.c* (where "s" stays for Server), while the BLE microcontroller of the External uses *amtc.c* (where the first "c" stays for Client). These three codes, together with the necessary functions, belong to *ble_app_att_mtu_throughput* code.

As the following sections suggest, the final result is obtained by working on the singular connections over time: the BLE connection between two microcontrollers, then the Implanted SPI protocol and finally on the External SPI connection.

3.1 BLE Server - BLE Client communication

Each microcontroller runs the same code in this section:
ble_app_att_mtu_throughput. This code is needed for the final result of this project, but it is a Nordic example code, and its scope was to calculate the throughput of BLE communications.

3.1.1 *main.c*

main.c is the code the two BLE microcontrollers share. It sets up and manages BLE communication between two boards: one acting as a Tester and the other as a Responder. The goal is to measure the data transfer rate. The program initializes the BLE stack, GATT, and GAP parameters and configures connection parameters for optimal throughput.

The program starts including libraries, defining LEDs and buttons and initializing connection parameters for optimal throughput:

```c
#define CONN_INTERVAL_DEFAULT           (uint16_t)(MSEC_TO_UNITS
    (7.5, UNIT_1_25_MS))    /**< Default connection interval used
    at connection establishment by central side. */

#define CONN_INTERVAL_MIN               (uint16_t)(MSEC_TO_UNITS
    (7.5, UNIT_1_25_MS))    /**< Minimum acceptable connection
    interval, in units of 1.25 ms. */
#define CONN_INTERVAL_MAX               (uint16_t)(MSEC_TO_UNITS
    (7.5, UNIT_1_25_MS))    /**< Maximum acceptable connection
    interval, in units of 1.25 ms. */
#define CONN_SUP_TIMEOUT                (uint16_t)(MSEC_TO_UNITS
    (4000,  UNIT_10_MS))    /**< Connection supervisory timeout (4
    seconds). */
#define SLAVE_LATENCY                   0
                                                  /**< Slave
    latency. */
```

Additional tests are conducted by changing the value of CONN_INTERVAL_DEFAULT and monitoring whether the throughput improves. In the frame of code above, CONN_INTERVAL_MIN and CONN_INTERVAL_MAX have the same value in order to ensure a CI of 7.5 ms since it is the value which ensures less latency, as explained in Bluetooth 4.2. If these two values were not equal, the CONN_INTERVAL_DEFAULT is not ensured, and so the CI could be a number between CONN_INTERVAL_MIN and CONN_INTERVAL_MAX.

The Tester (the Server) or Responder (the Client) board roles can be set using buttons. The Tester starts the throughput test while the Responder responds.

```c
static board_role_t volatile m_board_role = NOT_SELECTED;

case BOARD_TESTER_BUTTON:
    m_board_role = BOARD_TESTER;

case BOARD_DUMMY_BUTTON:
    m_board_role = BOARD_DUMMY;
    advertising_start(); // Starts advertising for connections.
    scan_start();        // Begins scanning for devices.
```

As highlighted in Bluetooth 4.2, in this part of the project, the DLE was carried out in order to obtain the maximum from the throughput. Here the lines of *sdk_config.c* where the DLE was set:

```
//<i> Requested BLE GAP data length to be negotiated.

#ifndef NRF_SDH_BLE_GAP_DATA_LENGTH
#define NRF_SDH_BLE_GAP_DATA_LENGTH 251      // It was 27
#endif
```

```
// <o> NRF_SDH_BLE_GATT_MAX_MTU_SIZE - Static maximum MTU size.

#ifndef NRF_SDH_BLE_GATT_MAX_MTU_SIZE
#define NRF_SDH_BLE_GATT_MAX_MTU_SIZE 247   // It was 23
#endif
```

To better comprehend the choice of these values, see in detail Fig. 1.6. It is possible to see that, with DLE, the Data Channel Payload is made of three parts: the L2CAP Header is made of 4 bytes, the ATT Header is made of 3 bytes, and ATT Data is made of 244 bytes, which is the maximum amount of attribute data that can be sent in a single over-the-air packet over BLE. So, the actual data length is 244 bytes since the ATT protocol adds a 3-byte header, and the link layer adds another 4-byte header on top of the attribute data. Indeed, the reason why NRF_SDH_BLE_GATT_MAX_MTU_SIZE is set to 247 is to make room for the 3-byte ATT header while still allowing sending the maximum 244 bytes of attribute data. According to the Bluetooth specification, an attribute can not be larger than 512 bytes (not including the 3-byte header). To send data larger than this limit, it must be split first into multiple packets. Therefore, although the theoretical maximum size for a GATT attribute is 512 bytes, practical limitations such as hardware constraints and buffer management often require smaller sizes. A commonly adopted size is 247 bytes, making the most efficient way to transmit data over BLE to split it into 244-byte chunks. This approach maximizes the capacity of BLE packets while minimizing overhead [23].

Then the test parameters as:

- NRF_SDH_BLE_GATT_MAX_MTU_SIZE, set at 247 byte and indicates the MTU size that defines the maximum payload size of GATT packets exchanged between the BLE devices;
- NRF_SDH_BLE_GAP_DATA_LENGTH, set at 251 byte and indicates the Maximum data length for BLE GAP layer and it is the sum of NRF_SDH_BLE_GATT_MAX_MTU_SIZE and L2CAP Header;
- BLE_GAP_PHY, set in rx and tx at BLE_GAP_PHY_2MBPS or alternatively at 1 Mbps (if not supported);
- CONN_INTERVAL_DEFAULT, set at 7.5 ms and is the default connection interval used at connection establishment by central side;
- CONN_INTERVAL_MIN, set at 7.5 ms and is the Minimum acceptable connection interval;
- CONN_INTERVAL_MAX, set at 7.5 ms and is the Maximum acceptable connection interval;

- SLAVE_LATENCY, set at 0, allows to not compromise the connection letting the peripheral to skip consecutive connection Events and not listen to the central at these connection events;
- CONN_SUP_TIMEOUT, set at 4 s, over this time the disconnection occurs;

are set as indicates the code below.

```c
static test_params_t m_test_params =
{
    .att_mtu                  = NRF_SDH_BLE_GATT_MAX_MTU_SIZE,
    .data_len                 = NRF_SDH_BLE_GAP_DATA_LENGTH,
    .conn_interval            = CONN_INTERVAL_DEFAULT,
    .conn_evt_len_ext_enabled = true,
    // Only symmetric PHYs are supported.
#if defined(S140)
    .phys.tx_phys             = BLE_GAP_PHY_2MBPS |
       BLE_GAP_PHY_1MBPS | BLE_GAP_PHY_CODED,
    .phys.rx_phys             = BLE_GAP_PHY_2MBPS |
       BLE_GAP_PHY_1MBPS | BLE_GAP_PHY_CODED,
#else
    .phys.tx_phys             = BLE_GAP_PHY_2MBPS |
       BLE_GAP_PHY_1MBPS,
    .phys.rx_phys             = BLE_GAP_PHY_2MBPS |
       BLE_GAP_PHY_1MBPS,
#endif
};


// Connection parameters requested for connection.
static ble_gap_conn_params_t m_conn_param =
{
    .min_conn_interval = CONN_INTERVAL_MIN,    // Minimum connection
        interval.
    .max_conn_interval = CONN_INTERVAL_MAX,    // Maximum connection
        interval.
    .slave_latency     = SLAVE_LATENCY,        // Slave latency.
    .conn_sup_timeout  = CONN_SUP_TIMEOUT      // Supervisory
       timeout.
};
```

Two functions in *main.c*, *amts_evt_handler()* and *amtc_evt_handler()*, recall *amts.c* and *amtc.c*.

The Server function:

- communicates if the Server board is ready with the event NRF_BLE_AMTS_EVT_NOTIF_ENABLED;
- checks if the notifications are disabled;
- communicates if 244 bytes are transferred when NRF_BLE_AMTS_EVT_TRANSFER_244B occurs;
- when the NRF_BLE_AMTS_EVT_TRANSFER_FINISHED occurs, calculates the throughput, keeping the time of the configuration using the functions

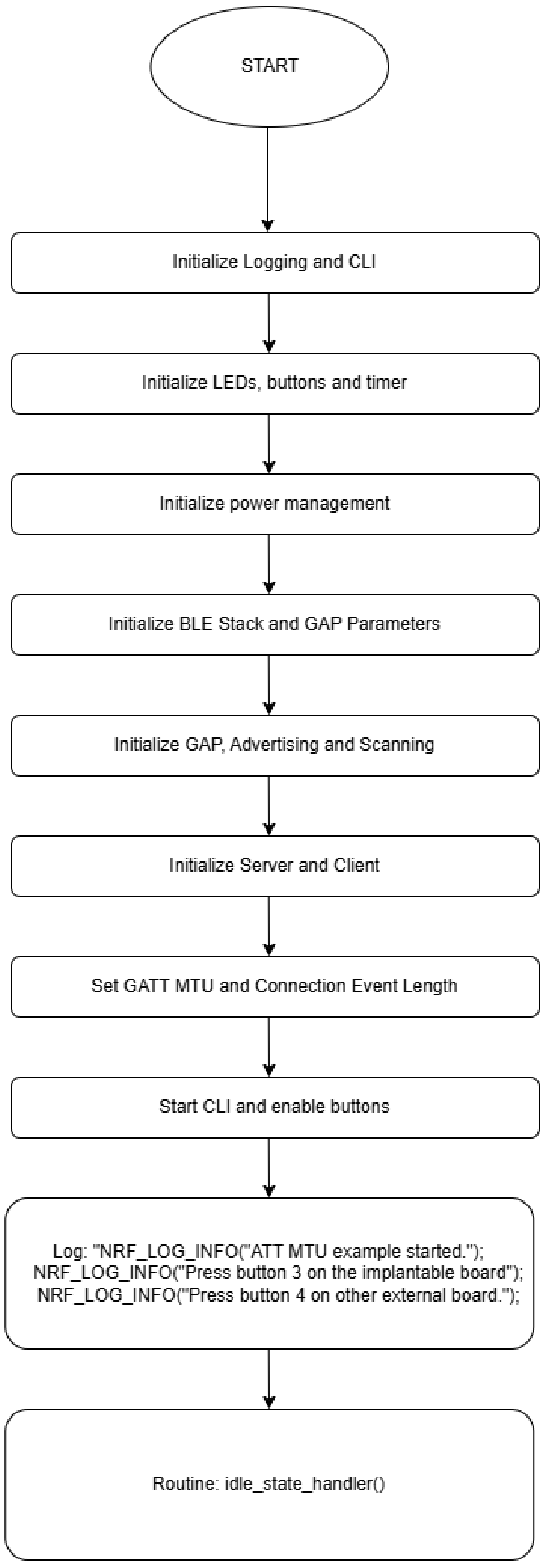
counter_get() and *counter_stop()* considering the number of bit transferred with the member *bytes_transfered_cnt* of the struct *nrf_ble_amts_evt_t*.

On the other hand, the Client function:

- ensures the client device finds and identifies the ATT MTU Throughput Service on the server device it is connected to when NRF_BLE_AMT_C_EVT_DISCOVERY_COMPLETE occurs;
- counts the bytes - then also the packets - received when NRF_BLE_AMT_C_EVT_NOTIFICATION is verified;
- communicates how many bytes of ATT payload received from the peer (the server) with NRF_BLE_AMT_C_EVT_RBC_READ_RSP.

Below is shown the code of the *amtc_evt_handler()* function.

This flowchart in Fig. 3.1 presents a global view of *main.c* above explained:


START
Initialize Logging and CLI
Initialize LEDs, buttons and timer
Initialize power management
Initialize BLE Stack and GAP Parameters
Initialize GAP, Advertising and Scanning
Initialize Server and Client
Set GATT MTU and Connection Event Length
Start CLI and enable buttons
Routine: idle_state_handler()

```
Log: "NRF_LOG_INFO("ATT MTU example started.");
NRF_LOG_INFO("Press button 3 on the implantable board");
NRF_LOG_INFO("Press button 4 on other external board.");
```

Figure 3.1: Flowchart of main.c .

The last block of the flowchart is described as routine. This means that the function is called in the *int_main()* function of *main.c*. The flowchart of this routine is shown in Fig. 3.2:

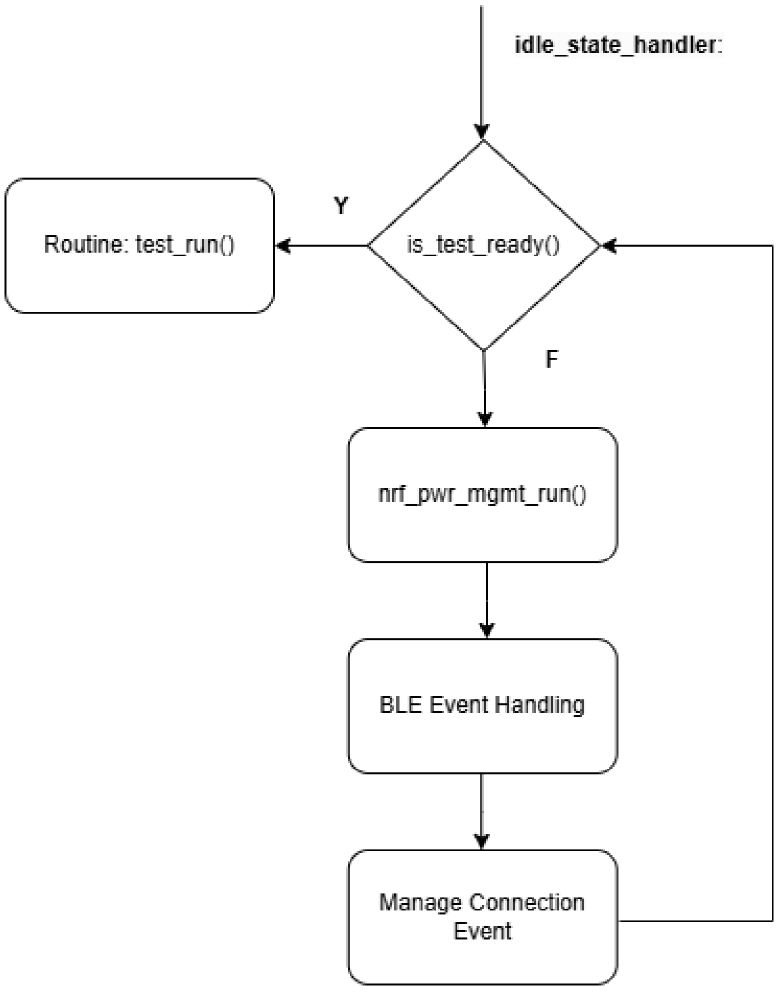


Figure 3.2: Flowchart of idle_state_handler() routine.

Also in the previous flowchart it is possible to see a routine, in this case is the *test_run()* flowchart in Fig. 3.3:

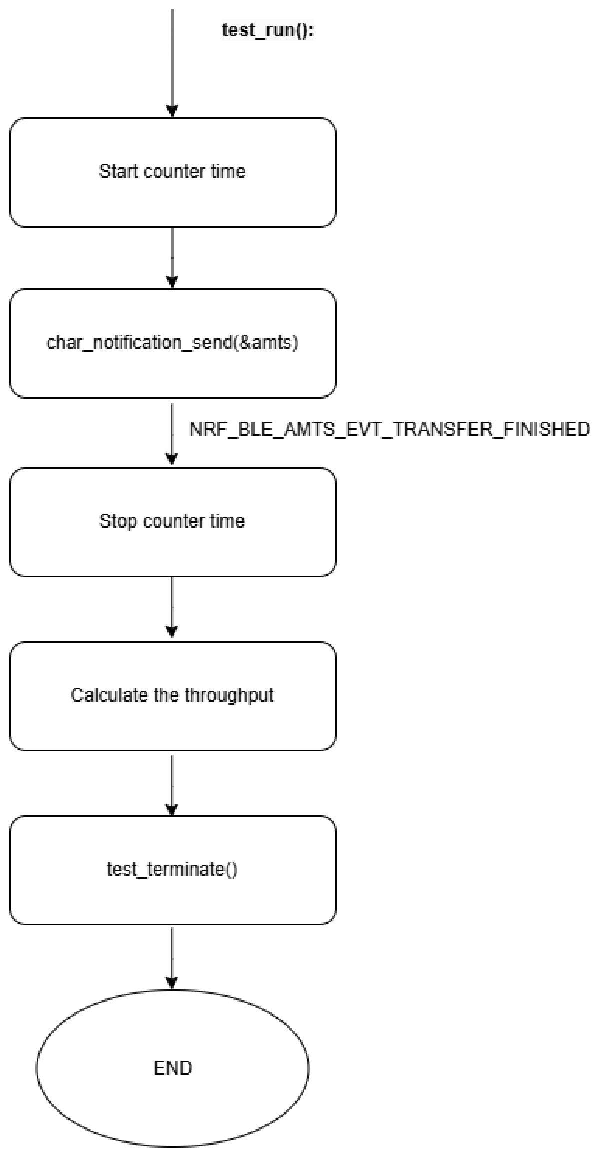


Figure 3.3: Flowchart of test_run routine.

As it is possible to see in the flowchart in Fig. 3.3, the *amts.c* program is called from the *main.c*.

3.1.2 *amts.c*

amts.c is the code which manages the Server's functions.

It is linked with the principal code *main.c* through many functions. The first analyzed is *char_notification_send()*:

```
static void char_notification_send(nrf_ble_amts_t * p_ctx)
```

```
{
    uint8_t             data[256];
    uint16_t            payload_len = p_ctx->max_payload_len;
    nrf_ble_amts_evt_t  evt;

    if (p_ctx->bytes_sent >= AMT_BYTE_TRANSFER_CNT)
    {
        evt.bytes_transfered_cnt = p_ctx->bytes_sent;
        evt.evt_type             =
           NRF_BLE_AMTS_EVT_TRANSFER_FINISHED;

        p_ctx->evt_handler(evt);

        p_ctx->busy        = false;
        p_ctx->bytes_sent  = 0;
        p_ctx->kbytes_sent = 0;

        return;
    }

     // Preparing packets with values from 0 to 243
     for (uint16_t i = 0; i < payload_len; i++) {
         data[i] = i % 244;
    }

    ble_gatts_hvx_params_t const hvx_param =
    {
        .type   = BLE_GATT_HVX_NOTIFICATION,
        .handle = p_ctx->amts_char_handles.value_handle,
        .p_data = data,
        .p_len  = &payload_len,
    };

    uint32_t err_code = NRF_SUCCESS;
    while (err_code == NRF_SUCCESS)
    {
        (void)uint32_encode(p_ctx->bytes_sent, data);

        err_code = sd_ble_gatts_hvx(p_ctx->conn_handle, &hvx_param)
           ;

        if (err_code == NRF_ERROR_RESOURCES)
        {
            // Wait for BLE_GATTS_EVT_HVN_TX_COMPLETE.
            p_ctx->busy = true;
            break;
        }
        else if (err_code != NRF_SUCCESS)
        {
            NRF_LOG_ERROR("sd_ble_gatts_hvx()␣failed:␣0x%x",
                err_code);
        }

        p_ctx->bytes_sent += payload_len;

        if (p_ctx->kbytes_sent != (p_ctx->bytes_sent / 244))
        {
            p_ctx->kbytes_sent = (p_ctx->bytes_sent / 244);

            evt.evt_type              =
                NRF_BLE_AMTS_EVT_TRANSFER_244B;
            evt.bytes_transfered_cnt = p_ctx->bytes_sent;
            p_ctx->evt_handler(evt);
        }
    }
}
```

This function sends 244-byte packets filled with value from 0 to 243 (it is also verified using Wireshark as confirm). It is also necessary clarify the role of AMT_BYTE_TRANSFER_CNT: it is declared in *amt.h* (the header of *amts.c* and *amtc.c*) and represents the amount of byte to be transferred. So, to transfer five 244-byte packets of data, it needs to be configured in this way:

```
#define AMT_BYTE_TRANSFER_CNT                   (5 * 244)
```

At each transfer the NRF_BLE_AMTS_EVT_TRANSFER_244B occurs and if meanwhile bytes_sent $\geq$ AMT_BYTE_TRANSFER_CNT so
NRF_BLE_AMTS_EVT_TRANSFER_FINISHED occurs and test_terminate() is called.

In the code above, as in the frames of code in the next sections, the function which sends data via BLE is *sd_ble_gatts_hvx()*.

The *char_notification_send()* function is called in two main point:

- *on_tx_complete()*, called in *amts.c* when BLE_GATTS_EVT_HVN_TX_COMPLETE event occurs then the program sends data when a notify is transferred;
- *nrf_ble_amts_notif_spam()*, called in *main.c* by *test_run()*, as said in *main.c*, to access to the *amts.c* and begin the transfer data.

An other function of *amts.c* called in *main.c* is *nrf_ble_amts_init()*. This function initializes the ATT MTU Throughput Service (AMTS) on the BLE server. To initalize AMTS it is necessary adding the service and defining its characteristics that facilitate data transfer and performance monitoring. Then, at first is defined the service base

```
ble_uuid128_t base_uuid = {SERVICE_UUID_BASE};

err_code = sd_ble_uuid_vs_add(&base_uuid, &(p_ctx->uuid_type));
APP_ERROR_CHECK(err_code);
```

next, is added the AMT service to the GATT server

```
ble_uuid.type = p_ctx->uuid_type;
ble_uuid.uuid = AMT_SERVICE_UUID;

// Add service.
err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &
    ble_uuid, &service_handle);
APP_ERROR_CHECK(err_code);
```

The BLE_GATTS_SRVC_TYPE_PRIMARY indicates that this is a primary service. At this point the AMTS characteristic is added in order to handles data transfer, sending data to the client to measure throughput

```
ble_add_char_params_t amt_params;
memset(&amt_params, 0, sizeof(amt_params));

amt_params.uuid              = AMTS_CHAR_UUID;
amt_params.uuid_type         = p_ctx->uuid_type;
amt_params.max_len           = NRF_SDH_BLE_GATT_MAX_MTU_SIZE;
amt_params.char_props.notify = 1;
amt_params.cccd_write_access = SEC_OPEN;
amt_params.is_var_len        = 1;

err_code = characteristic_add(service_handle, &amt_params, &(p_ctx
    ->amts_char_handles));
APP_ERROR_CHECK(err_code);
```

AMTS_CHAR_UUID contains Universally Unique Identifier (UUID), a 128-bit value "universally unique identifier that is guaranteed to be unique across all space and all time"[24]. It defines the primary data transfer characteristic within the AMT service. It uses BLE notifications to send data from the server to the client allowing the server to push data to the client without a read request, differently to the writing operation.

Finally, the AMT Received Bytes Count Characteristic is added. It keeps track of the number of bytes received by the server through AMT_RCV_BYTES_CNT_CHAR_UUID.

Also *amts.c*, as *main.c*, has a ble_event_handler, called *nrf_ble_amts_on_ble_evt()* that manages the most important events in the code:

```
void nrf_ble_amts_on_ble_evt(ble_evt_t const * p_ble_evt, void *
    p_context)
{
nrf_ble_amts_t * p_ctx = (nrf_ble_amts_t *)p_context;

switch (p_ble_evt->header.evt_id)
{
        case BLE_GAP_EVT_CONNECTED:
            on_connect(p_ctx, p_ble_evt);
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            on_disconnect(p_ctx, p_ble_evt);
            break;

        case BLE_GATTS_EVT_WRITE:
            on_write(p_ctx, p_ble_evt);
            break;

        case BLE_GATTS_EVT_HVN_TX_COMPLETE:
            on_tx_complete(p_ctx);
            break;

        default:
            break;
    }
}
```

As the nrf_ble_amts_on_ble_evt() shows, also the on_write() is a main function of amts.c . It allows writing to the CCCD in order to enable or disabled notification.

The Fig. 3.4 shows the amts.c flowchart.

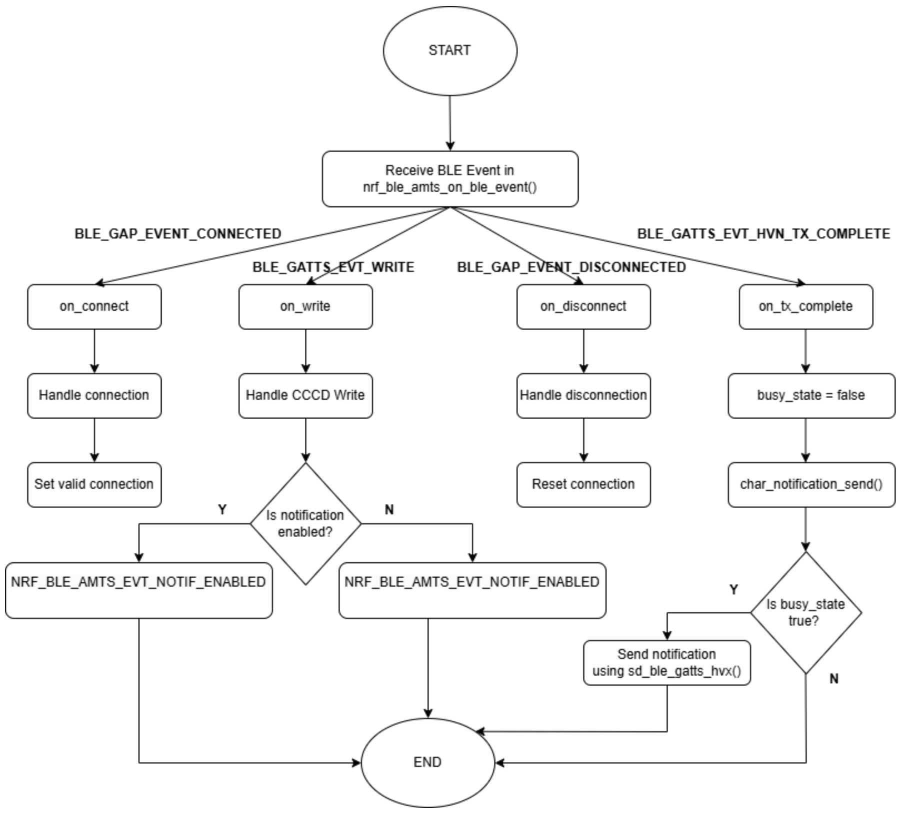


Figure 3.4: Flowchart of amts.c .

3.1.3 *amtc.c*

In order to better understand *amtc.c*, it is essential to start analyzing it from the ble handler *nrf_ble_amtc_on_ble_evt()*

```c
void nrf_ble_amtc_on_ble_evt(ble_evt_t const * p_ble_evt, void *
    p_context)
{
    nrf_ble_amtc_t * p_ctx = (nrf_ble_amtc_t *)p_context;

    if ((p_ctx == NULL) || (p_ble_evt == NULL))
    {
        return;
    }

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GATTC_EVT_HVX:
            on_hvx(p_ctx, p_ble_evt);
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            on_disconnected(p_ctx, p_ble_evt);
            break;

        case BLE_GATTC_EVT_WRITE_RSP:
            on_write_response(p_ctx, p_ble_evt);
            break;

        case BLE_GATTC_EVT_READ_RSP:
            on_read_response(p_ctx, p_ble_evt);
            break;

        default:
            break;
    }
}
```

The BLE_GATTC_EVT_HVX is the event triggered when a Notification event occurs. It is definible as the "gate" thank to which is possible to control data arriving. In fact, this part of the code was modified to include data_control for incoming data.

```c
static void on_hvx(nrf_ble_amtc_t * p_ctx, ble_evt_t const *
    p_ble_evt)
{
    // Check if the event if on the link for this instance
    if (p_ctx->conn_handle != p_ble_evt->evt.gattc_evt.conn_handle)
    {
        return;
    }

    for (uint16_t i = 0; i < 244; i++) {
        data_control[i] = i % 244;  // Filling the buffer with values from 0 to 243
    }

    // Comparing received data with data_control
    bool match = true;
    for (uint16_t i = 0; i < payload_len; i++) {
        if (p_ble_evt->evt.gattc_evt.params.hvx.data[i] != data_control[i]) {
            NRF_LOG_ERROR("Mismatch␣at␣byte␣%d:␣expected␣%d,␣got␣%d", i, data_control[i], p_ble_evt->evt.gattc_evt.params.hvx.data[i]);
            match = false;
            break;
        }

    }
    if (match) {
        NRF_LOG_INFO("Data␣received␣matches␣data_control.");
    } else {
        NRF_LOG_ERROR("Data␣received␣does␣not␣match␣data_control.");
    }

    p_ctx->evt_handler(p_ctx, &amt_c_evt);

}
```

To compare data_control with the input data it's essential to know the expected data in order to fill the array with these.

Going back to *nrf_ble_amtc_on_ble_evt()*, BLE_GAP_EVT_DISCONNECTED and BLE_GATTC_EVT_WRITE_RSP are similar to what occur in *amts.c* and, calling the respective functions for the Client: *on_disconnected()* and *on_write response()*, do the same job.

Finally, when the BLE_GATTC_EVT_READ_RSP event is triggered, the *on_read_response()* is called. The BLE_GATTC_EVT_READ_RSP is the event indicating that a known number of received bytes notification has been received from the peer. This number of bytes correspond to the amount of data which trigger NRF_BLE_AMTS_EVT_TRANSFER_244B.

In Fig. 3.5, the flowchart of amtc.c is presented.

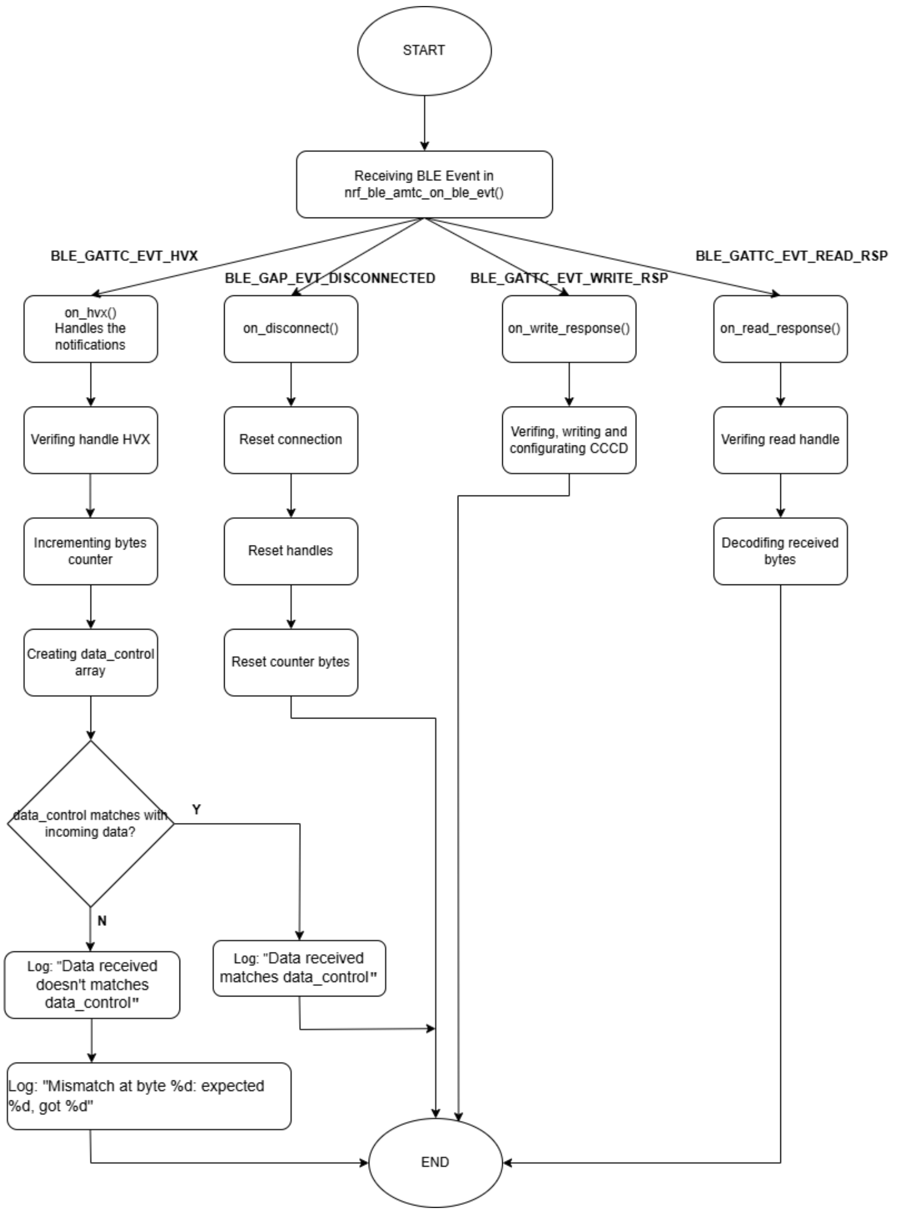

START
Receiving BLE Event in nrf_ble_amtc_on_ble_evt()
BLE_GATTC_EVT_HVX
BLE_GAP_EVT_DISCONNECTED
BLE_GATTC_EVT_WRITE_RSP
BLE_GATTC_EVT_READ_RSP
on_hvx() Handles the notifications
on_disconnect()
on_write_response()
on_read_response()
Verifing handle HVX
Reset connection
Verifing, writing and configurating CCCD
Verifing read handle
Incrementing bytes counter
Reset handles
Decodifing received bytes
Creating data_control array
Reset counter bytes
data_control matches with incoming data?
Y
N
Log: "Data received doesn't matches data_control"
Log: "Data received matches data_control"
Log: "Mismatch at byte %d: expected %d, got %d"
END

Figure 3.5: Flowchart of amtc.c .

3.2 SPI Implementation on BLE Server

In this section the code that allows the integration of SPI protocol on BLE Server, *spi.c* will be presented. In this way, data will not be generated in the amts.c, as explained in the section BLE Server - BLE Client, but data come from the SPI Master (SPIM) and it will be received via SPI by the BLE Server that now is also callable SPI Slave (SPIS).

In the next subsection the *amts.c* is slightly modified in order to set the SPI channel.

3.2.1 *amts.c* with SPI Implementation

To set a communication based on SPI protocol, it's needed to define an SPI istance:

```
#define SPIS_INSTANCE 1
```

It must be different from the SPI instance of the SPI Master.

In order to manage data via SPI it is required an SPI handler and an SPI initialization.

```
// SPI event handler
void spis_event_handler(nrf_drv_spis_event_t event) {
    if (event.evt_type == NRF_DRV_SPIS_XFER_DONE) {
        spis_xfer_done = true;
    }
}
```

In a system where synchronization between SPI devices is crucial, this handler allows the detection of the moment the data is ready to be processed or the moment a new transfer can be initiated. When the event NRF_DRV_SPIS_XFER_DONE occurs (indicating that an SPI transfer has successfully completed), the system can respond appropriately by setting the global variable spis_xfer_done to true (before initializated to false). This variable will subsequently be referenced in another code segment, which will be discussed later in this section.

```
// SPI initialization
void spis_init(void) {
    nrf_drv_spis_config_t spis_config = NRF_DRV_SPIS_DEFAULT_CONFIG
       ;
    spis_config.csn_pin = APP_SPIS_CS_PIN;
    spis_config.miso_pin = APP_SPIS_MISO_PIN;
    spis_config.mosi_pin = APP_SPIS_MOSI_PIN;
    spis_config.sck_pin = APP_SPIS_SCK_PIN;
    spis_config.mode = NRF_SPIS_MODE_0;

    APP_ERROR_CHECK(nrf_drv_spis_init(&spis, &spis_config,
       spis_event_handler));
}
```

This function configures and initializes the SPIS peripheral of the Nordic microcontroller. This is achieved through the members of the nrf_drv_spis_config_t structure, which contains the configuration parameters for the SPI Slave module.

At this point, the microcontroller is initialized and it will be ready to receive or transmit data as an SPIS with the SPI Master. The correct configuration of the pins and their mode is essential to ensure reliable and consistent SPI communication. Indeed, as shown in the code snippets, MOSI, MISO, CS and SCLK are initialized with values set in sdk_config.h according to the pin selected for SPI communication. In addition, the SPI communication mode is set to mode 0, which specifies a clock polarity (CPOL) of 0 (the clock is low when idle) and a clock phase (CPHA) of 0 (data is sampled on the rising edge of the clock).

In order to "open" the SPI channel it is necessary to declare *spis_init()* in *nrf_ble_amts_init()*.

Another aspect to take into account is the handling of data in the *amts.c* incoming via SPI. To better accomplish this task, it was preferred inglobe the function *char_notification_send()* in nrf_ble_amts_notif_spam() (in order to simplify the code) and to modify it as shown in the frame of code below:

```c
void nrf_ble_amts_notif_spam(nrf_ble_amts_t *p_ctx) {
    uint32_t err_code;
    uint32_t packets_received = 0;
    uint32_t total_bytes_received = 0;

    // Initializing buffers
    uint8_t *active_buf = buffer_1;
    uint8_t *passive_buf = buffer_2;

    // Receiving first SPI data in the active_buf before the while loop
    err_code = nrf_drv_spis_buffers_set(&spis, NULL, 0, active_buf, MAX_PACKET_SIZE);
    if (err_code != NRF_SUCCESS) {
        NRF_LOG_ERROR("Error␣in␣the␣setting␣of␣SPI␣buffer:␣0x%x", err_code);
        return;
    }

    // Waiting for the initial SPI transfer to be completed
    while (!spis_xfer_done) {
        __WFE(); // Waiting for the SPI transfer event to be completed
    }
    spis_xfer_done = false; // Resetting the flag

    while (packets_received < NUM_PACKETS) {
        // Setting the SPI buffer to receive data in the passive_buf
        err_code = nrf_drv_spis_buffers_set(&spis, NULL, 0, passive_buf, MAX_PACKET_SIZE);
        if (err_code != NRF_SUCCESS) {
            NRF_LOG_ERROR("Error␣in␣the␣setting␣of␣the␣SPI␣buffer:␣0x%x", err_code);
            break;
        }

        // Sending data from the active_buffer via BLE if notifications are enabled
        // Sending data in the active_buf via BLE if notify are enabled
        if (p_ctx->conn_handle != BLE_CONN_HANDLE_INVALID && is_ble_notification_enabled) {
            ble_gatts_hvx_params_t hvx_params = {
                .type = BLE_GATT_HVX_NOTIFICATION,
                .handle = p_ctx->amts_char_handles.value_handle,
                .p_data = active_buf,
                .p_len = &length
            };

            do {
                err_code = sd_ble_gatts_hvx(p_ctx->conn_handle, &hvx_params);
                if (err_code == NRF_ERROR_RESOURCES) {
                    // Attendi l'evento BLE_GATTS_EVT_HVN_TX_COMPLETE per liberare risorse
                    p_ctx->busy = true;
                }
            } while (err_code == NRF_ERROR_RESOURCES);

            if (err_code != NRF_SUCCESS) {
                NRF_LOG_ERROR("Errore␣during␣the␣BLE␣sending:␣0x%x", err_code);
                break;
            }
        }

        // Waiting for the SPI receiving to be completed
        while (!spis_xfer_done) {
            __WFE(); // Waiting for the SPI transfer to be completed
        }
        spis_xfer_done = false;

        // Swap the buffers: the passive become active and vice versa
        uint8_t *temp = active_buf;
        active_buf = passive_buf;
        passive_buf = temp;


    }

    NRF_LOG_INFO("All␣the␣%d␣SPI␣packets␣received␣and␣send␣via␣BLE.␣Bytes␣received:␣%d", PACKETS, total_bytes_received);

    // Invia l'evento di completamento trasferimento
    nrf_ble_amts_evt_t evt = {
        .evt_type = NRF_BLE_AMTS_EVT_TRANSFER_FINISHED,
        .bytes_transfered_cnt = total_bytes_received,

    };
}
```

In this code it is possible to see two buffers: active_buf and passive_buf. It is necessary to declare these two buffers because are used as pointer to buffer_1 and buffer_2 which are static buffers initialized in the firsts line of *amts.c*. Another reason to use two buffers is managing SPI data incoming and BLE data outcoming: efficiency is asked and just one buffer can't fulfil the request. In respect to the original *char_notification_send()*, the condition of the loop has been modified: now the code send data by packet and not by byte. It is been observed that, by opting for this solution, the behaviour system in data trasmission results better. Indeed, before the condition was

```
    if (p_ctx->bytes_sent >= AMT_BYTE_TRANSFER_CNT)
```

where, as declared in the previous section, AMT_BYTE_TRANSFER_CNT represents the amount of data to send in byte. Now the condition is:

```
    while (packets_sent < PACKETS)
```

where PACKETS is another amt.h added variable useful to calculate the packets to send from AMT_BYTE_TRANSFER_CNT. It is declared in this way:

```
    #define PACKETS AMT_BYTE_TRANSFER_CNT/244
```

The flowchart of the previous paragraph in Fig. 3.4 is evolved in the scheme in Fig. 3.6

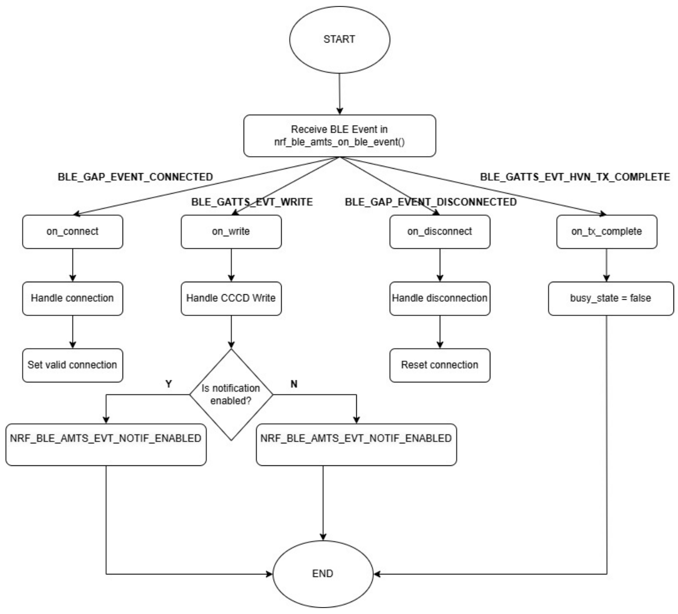


Figure 3.6: Flowchart of amts.c after SPI implementation on BLE Server.

As it possible to see from the previous flowchart, *nrf_ble_amts_notif_spam()* is not present because it is called in *test_run()* of main.c and since *nrf_ble_amts_notif_spam()* do also *char_notification_send()*, the program and flowchart are slimmed. Here in Fig. 3.7 the flowchart of *nrf_ble_amts_notif_spam()* that works meanwhile the rest of *amts.c* runs.

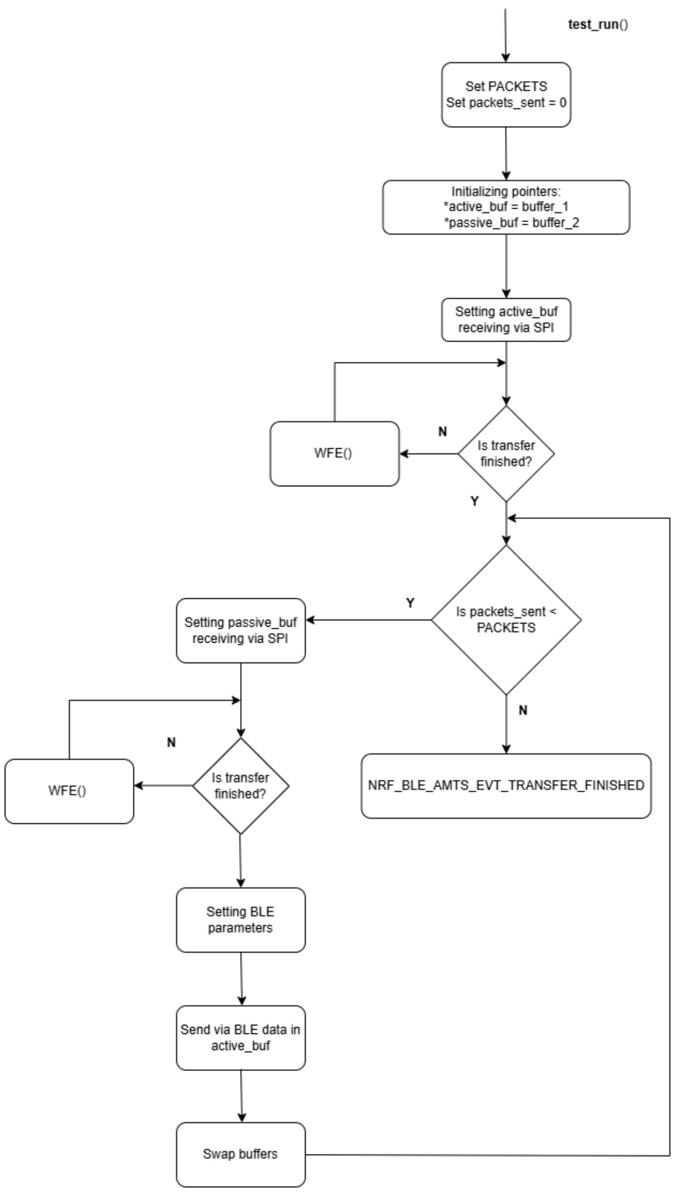


Figure 3.7: Flowchart of *nrf_ble_amts_notif_spam()*.

3.2.2 spi.c

Starting from the *spi.c* example of the Nordic SDK, it was necessary to slightly modify it in order to send via SPI a number of packets decided a priori.

As the SPIS, also the SPIM, has a proper SPI instance:

```
#define SPI_INSTANCE  0
```

As explained in amts.c with SPI Implementation section, it must be different from SPIS instance.

The code is composed by an *spi_event_handler()* and an *spi_init()*, as each code of SPIM or SPIS, to configure and initialize SPI protocol. Since this board performs as SPI Master, the *spi_init()* comprehends also the configuration of the frequency of the clock:

```
spi_config.frequency = NRF_SPI_FREQ_8M;
```

In this case this frequency is set at 8 MHz, in order to exploit the maximum data rate from the SPIM to compensate slownessof the throughput caused by the BLE congestion.

Then, the code includes the *int_main()* function that recalls the leds initialization, the *spi_init()*. This is achieved with a while loop which manages the outcoming data:

```
while(packet < PACKETS)
    {
        // Initializing the tx buffer with values from 0 to 243
        for (uint8_t i = 0; i < 244; i++)
        {
            m_tx_buf[i] = cnt++;
        }

        APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, m_tx_buf,
           m_length, NULL, 0)); // Starting the SPI transfer

        while (!spi_xfer_done) // Waiting for completed transfer
        {
            __WFE();           // This instruction sets the CPU to
               wait for an event
        }
        spi_xfer_done = false;          // Resetting the transfer
           flag

        packet++;

        bsp_board_led_invert(BSP_BOARD_LED_0);  // Inverting the
           led state

        my_nrf_delay_ms(18);
    }
```

At first, in this code, it is possible to notice data generation: it is the same operation that *amts.c* has provided when SPI protocol wasn't implemented. In this case the m_tx_buf array will be filled. At this point this array will be sent through the function *nrf_drv_spi_transfer()* that is the corrispective of the *nrf_drv_spis_buffers_set()* shown in the amts.c: the first function is used by SPI to send and receive data via SPI (in this case just to send since the rx elements are set to *NULL* and *0*), the second one is used BY spis to send and receive data via SPI (in this case just for receiving because the tx elements are set to *NULL* and *0*).

The WFE loop ensures data will be sent correctly while the flag spi_xfer_done is false.

Using Wireshark, it was observed that the packets arrived to Client were not consecutive. This leads to add a delay at the end of the while loop to manage the sending of packets better. As the function shows, the delay is not handled by *nrf_delay_ms()* (function of the *delay.h* of SDK) but by *my_nrf_delay_ms()*. The first one takes in input milliseconds values, the second one, a clone function of the *nrf_delay_ms()*, takes the argument and multiplies it for 0.1 ms. *nrf_delay_ms()* takes just integer value of delay, but with *my_nrf_delay_ms()*, using tenths of milliseconds as argument, is possible saving some milliseconds fraction enhancing the performance.

Before the end of the *int_main()* function, another WFE loop is required in order to keep the program always active.

```
while (1)      // Infinite loop to keep the program active
    {
        __WFE();
    }
```

In Fig. 3.8 is shown the flowchart of the code which run on the Implantable SPIM.

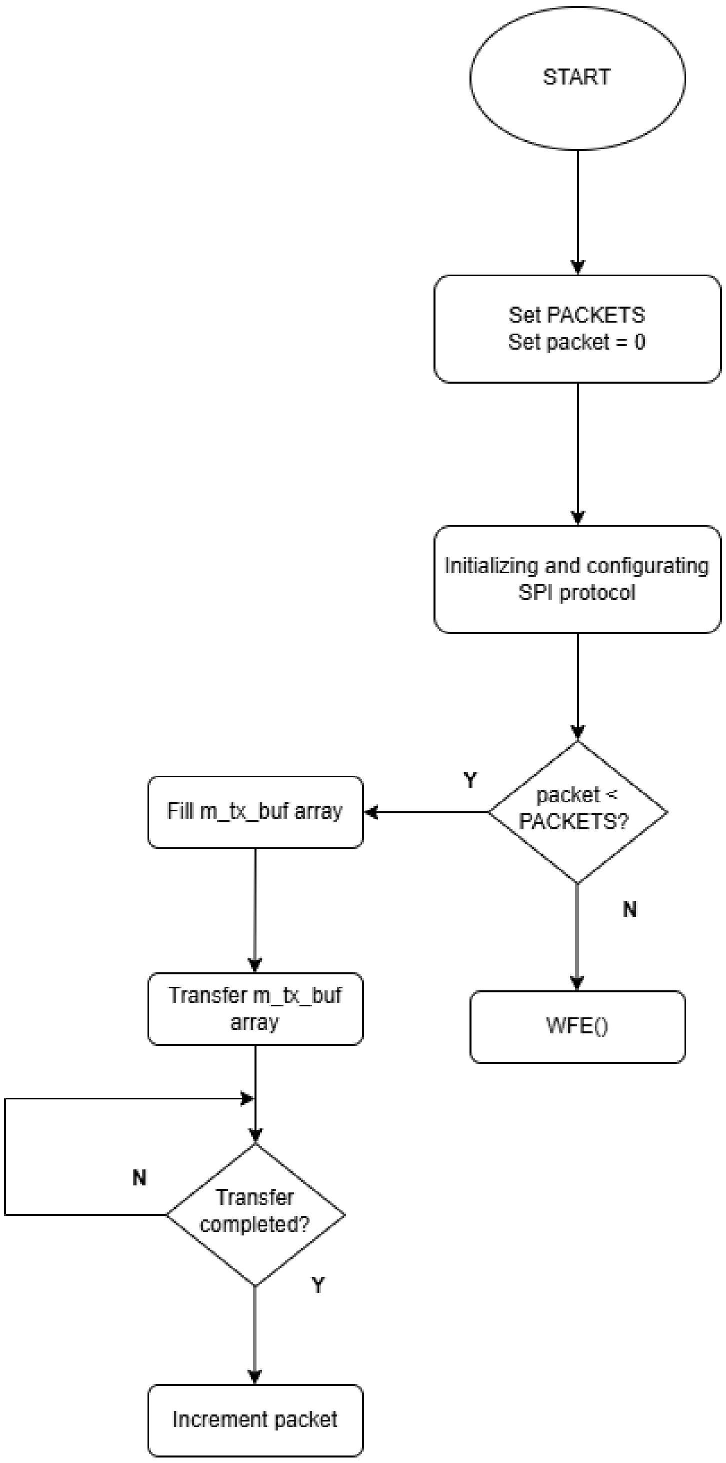


Figure 3.8: Flowchart of spi.c .

3.2.3 amtc.c generating responses

This part of the work *amtc.c* has been modified again. In this case, the Client generates data internally (in the sense that these data don't come via SPI). It sends a response to the Server at a specified frequency. For instance, in this code, four responses of 244 bytes are sent just when all the decided number of packets from the Server are received:

```c
static void on_hvx(nrf_ble_amtc_t *p_ctx, ble_evt_t const *
    p_ble_evt)
{
    if (p_ctx->conn_handle != p_ble_evt->evt.gattc_evt.conn_handle)
    {
        return;  // Exit if the actual connection isn't checked
    }

    if (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_ctx->
        peer_db.amt_handle)
    {
        received_packets++;

        // Response consists in the last four packets received
            adding 3 to each byte
        for (uint8_t i = 0; i < RESPONSE_SIZE; i++)
        {
            response_data[i] = p_ble_evt->evt.gattc_evt.params.hvx.
                data[i] + 3;
        }

        // Sending response packets
        if (received_packets >= PACKETS)
        {
            send_responses(p_ctx);  // Sending response packets
            received_packets = 0;   // Resetting the counter
        }
    }
}
```

The frequency of the response sending is modified in the last if condition. In the same if statement, is called back *send_responses()* function:

```c
static void send_responses(nrf_ble_amtc_t *p_ctx)
{
    for (uint16_t i = 0; i < NUM_RESPONSES; i++)
    {
        // If the ble_tx_complete flag is true, it is possible send
            a new packet
        if (ble_tx_complete)
        {
            if (p_ctx->conn_handle == BLE_CONN_HANDLE_INVALID)
            {
                NRF_LOG_ERROR("Valid␣connection.");
                return NRF_ERROR_INVALID_STATE;
            }

            ble_gatts_hvx_params_t hvx_params;
            memset(&hvx_params, 0, sizeof(hvx_params));
            hvx_params.handle = p_ctx->peer_db.amt_handle;
            hvx_params.type = BLE_GATT_HVX_NOTIFICATION;
            hvx_params.p_data = response_data;
            uint16_t length = RESPONSE_SIZE;
            hvx_params.p_len = &length;

            ret_code_t err_code = sd_ble_gatts_hvx(p_ctx->conn_handle, &hvx_params);

            // Handling error NRF_ERROR_RESOURCES
            if (err_code == NRF_ERROR_RESOURCES)
            {
                ble_tx_complete = false;  // Setting the flag to false until the  BLE_GATTS_EVT_HVN_TX_COMPLETE event isn't received
                NRF_LOG_INFO("Buffer␣full,␣waiting␣for␣BLE_GATTS_EVT_HVN_TX_COMPLETE.");
            }
            else if (err_code != NRF_SUCCESS)
            {
                NRF_LOG_ERROR("Error␣sending␣the␣notify:␣0x%x", err_code);
            }
            else
            {
                //NRF_LOG_INFO("Packet send succesfully.");
            }
        }
    }
}
```

This part of the code allows to send via BLE the *response_data* filled in *on_hvx()*.

In order to receive these data, in *amts.c* it is implemented a new version of *on_hvx*: it is added the case *BLE_GATTC_EVT_HVX*. Below the frame of code of *nrf_ble_amts_on_ble_evt*.

```
// BLE event manager
void nrf_ble_amts_on_ble_evt(ble_evt_t const *p_ble_evt, void *p_context) {
    nrf_ble_amts_t *p_ctx = (nrf_ble_amts_t *)p_context;

    switch (p_ble_evt->header.evt_id) {

        case BLE_GATTC_EVT_HVX:  // This event was added to the
           original amts.c because his scope wasn't to receive
           notifications
            on_hvx(p_ctx, p_ble_evt);
            break;
        ...
    }
}
```

The Fig. 3.9 depicts the flowchart of *amtc.c* that generates responses:

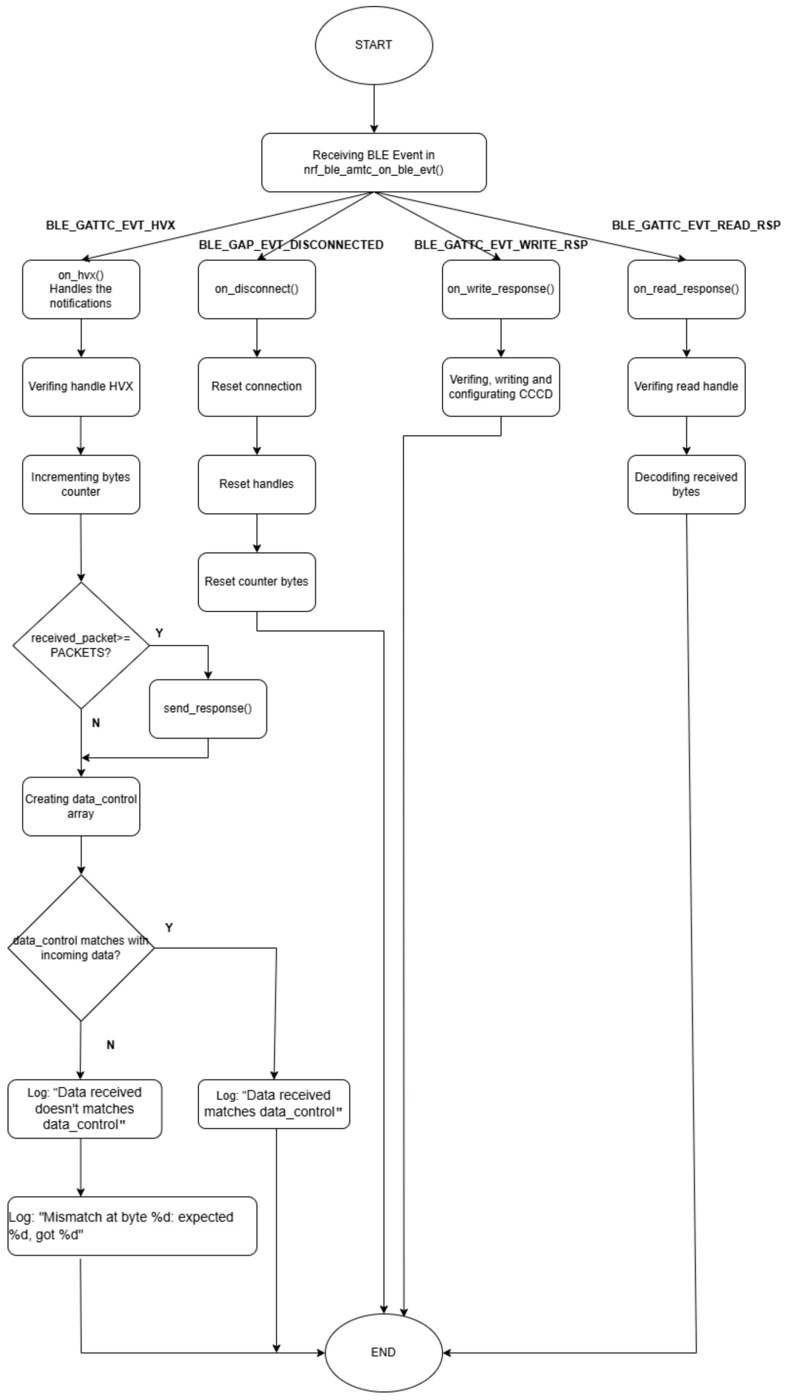

START
Receiving BLE Event in nrf_ble_amtc_on_ble_evt()
BLE_GATTC_EVT_HVX
BLE_GAP_EVT_DISCONNECTED
BLE_GATTC_EVT_WRITE_RSP
BLE_GATTC_EVT_READ_RSP
on_hvx() Handles the notifications
on_disconnect()
on_write_response()
on_read_response()
Verifing handle HVX
Reset connection
Verifing, writing and configurating CCCD
Verifing read handle
Incrementing bytes counter
Reset handles
Decodifing received bytes
Reset counter bytes
received_packet>= PACKETS?
Y
N
send_response()
Creating data_control array
data_control matches with incoming data?
Y
N
Log: "Data received doesn't matches data_control"
Log: "Data received matches data_control"
Log: "Mismatch at byte %d: expected %d, got %d"
END

Figure 3.9: Flowchart of *amtc.c* that generates responses.

3.3 BLE Server - BLE Client communication generating reply from SPI

An important difference between the Implantable module and the External module is the precence of the handshake based on GPIO: the Client will sends a response to the Server, after receiving a preset number of packets from the Server.

The number of packets to receive before send response is called FREQUENCY RESPONSE PACKETS. For example, if the total number of packets sent from the Server is 1000 and FREQUENCY RESPONSE PACKETS is set to 100, it means that every 100 packets received the Client will sends a response composed for example by four 244-bytes packets.

In Fig. 3.10 is shown the oscilloscope capture of the handhsake request. In blue the signal of the SPIM which receives the handshake request, in light blue the SPIS - also BLE Client - which sends responses.

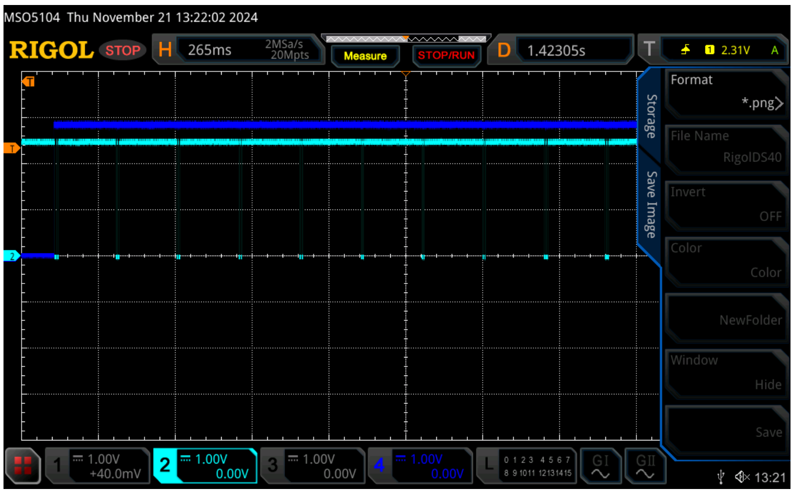


Figure 3.10: Oscilloscope capture which shown handshake request.

It is possible to see that the handshake request is arised just once, when *received_packets* is equal to Frequency Response Packets. In those case the SPIM signal is arised. Only if SPIM signal is high, when the module of received_packet with Frequency Response Packets is zero, the SPIS signal is inverted.

3.3.1 amtc.c after SPI implementation

In this section *amtc.c* will be analized. Its purpose is sending data as response with data from its SPI Master.

At the beginning the SPI protocol is initialized with *spis_event_handler()* and *spi_init()*. The pins chosen for the SPI communication are the same as the Implantable Module. The only change in spis initialization is the SPI mode of transfer, indeed the Mode 1 was set. It is characterised by CPOL = 0 and CPHA = 1: the clock signal is low when idle (CPOL = 0), and data is sampled on the falling edge of the clock (CPHA = 1) while being generated on the rising edge.

As presented in the previous chapters, when BLE_GATTC_EVT_HVX occurs, a notification arrives, and the function *on_hvx()* is triggered.
on_hvx() function has been modified:

```c
static void on_hvx(nrf_ble_amtc_t *p_ctx, ble_evt_t const *
    p_ble_evt)
{

    if (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_ctx->
        peer_db.amt_handle)
    {
        received_packets++;

        // Check if it's time to send responses
        if (received_packets == FREQUENCY_RESPONSE)
        {
            raise_handshake_request();  // Raise the handshake
                request

            // Ensure we handle all responses
            if (!responses_pending)
            {
                send_responses(p_ctx);
            }
            else
            {
                __WFE();
            }

        }
        else if (received_packets % FREQUENCY_RESPONSE == 0)
        {
            // Ensure we handle all responses
            if (!responses_pending)
            {
                send_responses(p_ctx);
            }
            else
            {
                __WFE();
            }
        }

        // Ensure we reset `received_packets` after sending all
            expected packets
        if (received_packets >= PACKETS)
        {
            received_packets = 0;  // Reset packet counter after
               all packets are processed

            // Creating and sending the transfer completed event
            nrf_ble_amtc_evt_t evt;
            evt.evt_type = NRF_BLE_AMTC_EVT_TRANSFER_FINISHED;
            evt.conn_handle = p_ctx->conn_handle;

            // Calling the event handler with the new event
            p_ctx->evt_handler(p_ctx, &evt);
            return;
        }
    }
}
```

As it is possible to see above, if the connection is verified, the counter *received_packets* increases by 1. Then, if the module of this counter with FREQUENCY RESPONSE PACKETS is 0, an handhsake request is sent.

```
void raise_handshake_request(void)
{
    nrf_gpio_pin_set(HANDSHAKE_PIN);  // Set the pin high to share
       it's free
    //NRF_LOG_INFO("Handshake request sent via GPIO.");
}
```

Then, if the flag *response_pending* is false, indicating that the program is sending any response, the *send_response()* function is triggered. Otherwhise, if the flag is true, the response via BLE is being transmitted.

To eliminate log in order to avoid the delay of the next events, the data control part of the code was eliminated, so the definitive flowchart of amtc.c is in Fig. 3.11.

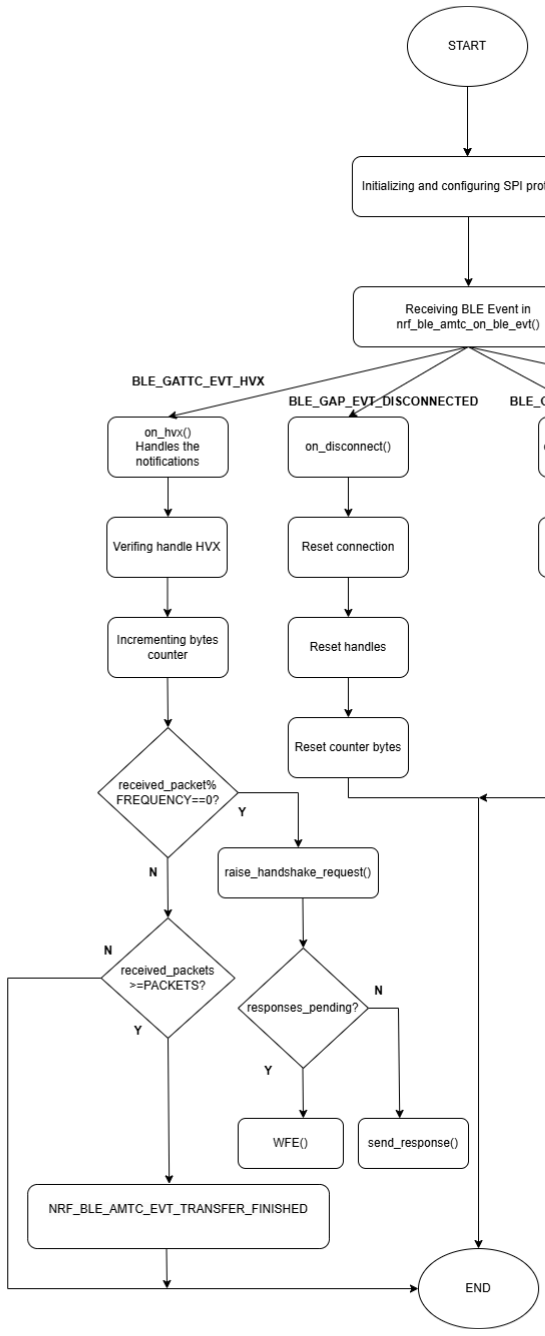

START
Initializing and configuring SPI protocol
Receiving BLE Event in nrf_ble_amtc_on_ble_evt()
BLE_GATTC_EVT_HVX
BLE_GAP_EVT_DISCONNECTED
BLE_GATTC_EVT_WRITE_RSP
BLE_GATTC_EVT_READ_RSP
on_hvx() Handles the notifications
on_disconnect()
on_write_response()
on_read_response()
Verifing handle HVX
Reset connection
Verifing, writing and configurating CCCD
Verifing read handle
Incrementing bytes counter
Reset handles
Decodifing received bytes
Reset counter bytes
received_packet% FREQUENCY==0?
Y
N
raise_handshake_request()
received_packets >=PACKETS?
N
Y
responses_pending?
N
Y
WFE()
send_response()
NRF_BLE_AMTC_EVT_TRANSFER_FINISHED
END

Figure 3.11: Definitive flowchart of amtc.c .

send_response() has the structure in Fig. 4.1:

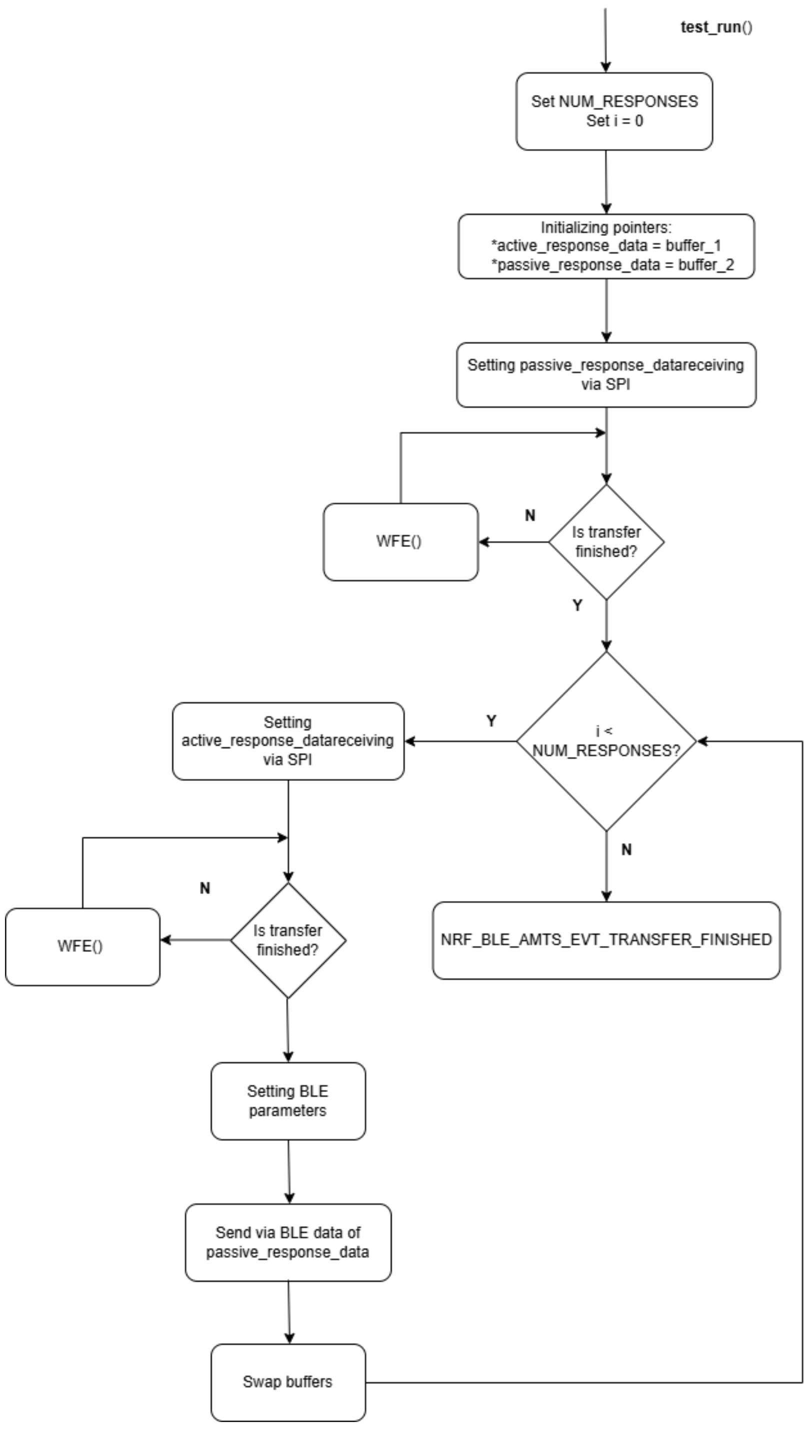

test_run()
Set NUM_RESPONSES
Set i = 0
Initializing pointers:
*active_response_data = buffer_1
*passive_response_data = buffer_2
Setting passive_response_datareceiving via SPI
Is transfer finished?
N
WFE()
Y
i < NUM_RESPONSES?
Y
Setting active_response_datareceiving via SPI
N
NRF_BLE_AMTS_EVT_TRANSFER_FINISHED
Is transfer finished?
N
WFE()
Setting BLE parameters
Send via BLE data of passive_response_data
Swap buffers

Figure 3.12: Flowchart of send_response().

As it is possible to appreciate in Fig. 4.1, this frame of code has the same structure of *nrf_ble_amts_notif_spam()*, indeed they have the same code.

3.3.2 spi_external.c

This code is inspired by *spi.c*. In addition to *spis_event_handler()* and *spis_init()* functions, there is also an *is_handshake_request_received()* function, since, as stated in the previous section amtc.c after SPI implementation, an handshake logic based on GPIO is needed.

```c
void spi_event_handler(nrf_drv_spi_evt_t const * p_event, void *
   p_context) {
    spi_xfer_done = true; // Set flag transfer complete
}

void spi_init(void){
// SPI Configuration
    nrf_drv_spi_config_t spi_config = NRF_DRV_SPI_DEFAULT_CONFIG;
    spi_config.ss_pin   = SPI_SS_PIN;
    spi_config.miso_pin = SPI_MISO_PIN;
    spi_config.mosi_pin = SPI_MOSI_PIN;
    spi_config.sck_pin  = SPI_SCK_PIN;
    spi_config.frequency = NRF_SPI_FREQ_8M;   // Frequency a 8 MHz
    spi_config.mode = NRF_SPI_MODE_1;

    // Initializing SPI Instance with the configuration
    APP_ERROR_CHECK(nrf_drv_spi_init(&spi, &spi_config,
       spi_event_handler, NULL));
}

/**
 * @brief Function to check the handshake
 * @return true if pin is set high, otherwise false.
 */
bool is_handshake_request_received() {
    return nrf_gpio_pin_read(HANDSHAKE_PIN) == 1;  // Read the
       state of pin handshake
}
```

So, the *int_main()* function is the same as the *int_main()* function of *spi.c()*, but it is regulated by the handshake:

```c
// Main loop for the hanshake handling and sending packets
    while (1) {
        // Checking if the handshake pin is set high
        if (is_handshake_request_received()) {

            // Update the TX buffer with new data
            for (uint8_t i = 0; i < MAX_PACKET_SIZE; i++) {
                m_tx_buf[i] = cnt++;  // Increases ‘cnt‘ to ensure
                    different data
            }

            // Sending the update packet to the Slave
            APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, m_tx_buf,
                MAX_PACKET_SIZE, NULL, 0));

            // Waiting for the transaction to complete
            while (!spi_xfer_done) {
                __WFE();  // Waiting For the Event
            }
            spi_xfer_done = false;  // Resetting the flag

            packet_num++;

            // Inverting the state of LED to indicate the sending
                of a packet
            bsp_board_led_invert(BSP_BOARD_LED_0);
        }
    }


    while (true) {  // Infinite loop to maintain the program always
        active
        __WFE();
    }
```

In Fig. 3.13, the flowchart of *spi_external.c* to better understand how it works.

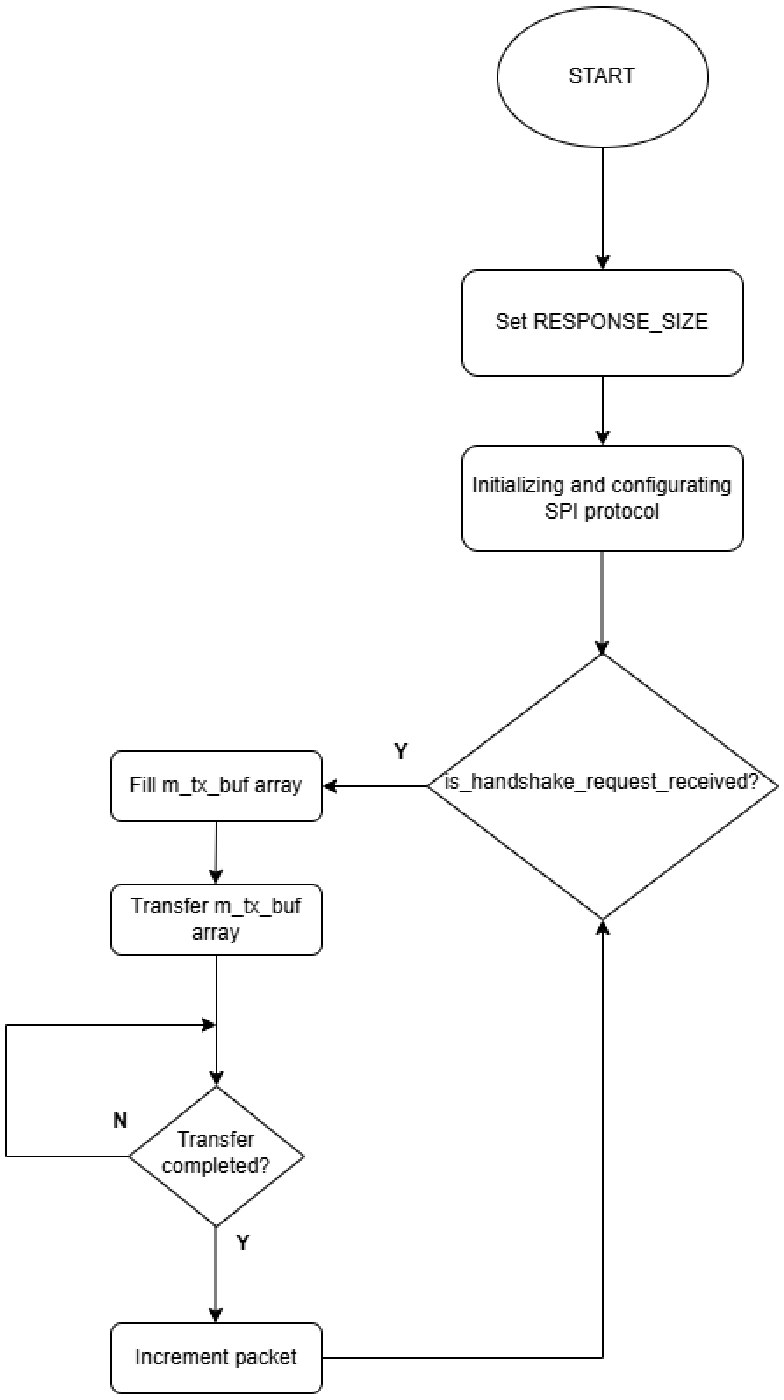


Figure 3.13: Flowchart of *spi_external.c*.

Chapter 4

Results

This paragraph shows the results obtained during the work and the tests conducted to reach them.

4.1 BLE Server - BLE Client

In this preliminary phase of the work, many tests were conducted to understand how the BLE protocol works.

In the BLE Server - BLE Client communication section, were explained the parameters, the scope, and how the *ble_app_att_mtu_throughput* works. The throughput log was exploited in order to visualize it. In this way, values of throughput has been collected to make Tab.1 and Tab.2.

The values in Tab.1 have been obtained by sending 100 packets of 244-byte size with 1M PHY and 2M PHY for each value of CI chosen. The values are shown as median and, in brackets, maximum and minimum values. In bold, the maximum values obtained. Data are sent via Notification.

CI (ms)	Throughput with 1M PHY (kbps)	Throughput with 2M PHY (kbps)
7.5	406.66 (406.66 – 424.34)	929.52 (1394.28 – 929.52)
10	**488 (488 – 488)**	**1220 (1394.28 – 976)**
50	235.18 (235.18 – 235.18)	368.3 (330.84 – 375.68)
100	146.76 (146.76 – 146.76)	179.08 (179.08 – 179.08)
200	84.13 (84.13 – 84.13)	93.84 (93.84 – 93.84)
500	36.83 (36.83 – 36.83)	38.57 (38.57 – 38.57)
1000	19.02 (19.02 – 19.02)	19.46 (19.48 – 19.46)

Table 4.1: Variation of throughput on basis of CI and PHY, sending ten packets of 244 bytes.

In Tab.2, as for Tab.1, the values of throughput are listed sending 1000 packets of 244-byte size with 1M PHY and 2M PHY, for each value of CI chosen. Data are also sent via notifications in this case.

CI (ms)	Throughput with 1M PHY (kbps)	Throughput with 2M PHY (kbps)
7.5	341.43 (341.67 – 341.43)	1047.21 (1049.46 – 1046.64)
10	354.78 (354.78 – 354.84)	985.85 (984.86 – 985.85)
50	**390.71 (391.1 – 390.55)**	**1297,87 (1303.97 – 1297.87)**
100	391.88 (391.88 – 391.81)	1290.15 (1291 – 1289.29)
200	386.22 (386.45 – 386.07)	1225.36 (1225.36 – 1224.59)
500	365.88 (365.95 – 365.88)	1039.4 (1039.4 – 1039.4)
1000	335.1 (335.1 – 335.1)	822.58 (822.58 – 822.58))

Table 4.2: Variation of throughput based on CI and PHY, sending 1000 packets of 244 bytes.

From these two necessary tables, it is possible to evaluate the number of bytes sent in one CI and find a trade-off to choose the correct parameters to meet the specifications.

The primary objective is to transmit all the data in as few intervals as possible. For example, in Tab.1, a CI of 10 ms was sufficient to obtain the best possible performance, but the same is not true for 1000 packets. By considering the system without any latency from other source, with a lower data load, a low CI is preferable as packets can be transmitted quickly, minimizing the intrinsic latency of BLE. A longer CI is more beneficial when the data load is high, as it allows more packets to accumulate and be sent in bulk, reducing overhead and improving efficiency.

4.2 SPI Implementation on BLE Server

In this project, SPI communication was used to facilitate data exchange between microcontrollers within both the internal and external modules, as SPI is considered to be a synchronous and fast protocol able to transfer data in a few microseconds or milliseconds. Although SPI is a fast wireless protocol, BLE has intrinsic latency associated with the CI and radio channel management. Depending on the CI chosen, bottlenecks can occur because data arrive quickly from the SPI Master, but has to wait for the next CE to be transmitted to the BLE Client. This delay increases with the CI: the longer the CI, the greater the likelihood of introducing more significant overall latency. A lower CI means a higher BLE update frequency and more immediate SPI data transmission, reducing overall latency.

This phenomenon was observed during the project. Consequently, when deciding to send via BLE 1000 packets arrived via SPI. In this case it was more advantageous to select a CI of 7.5 ms rather than 50 ms, as demonstrated in Tab.2.

The graph in Fig. 4.1 shows how SPI Implementation on the BLE Client impacts, considering SPI implementation on the BLE Server already implemented and not.

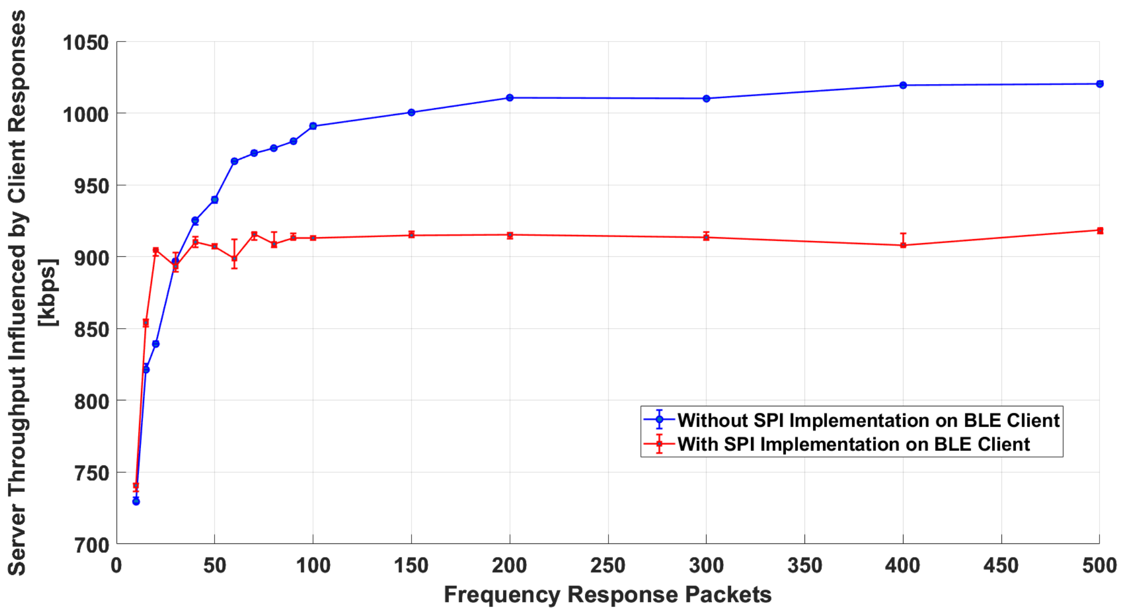


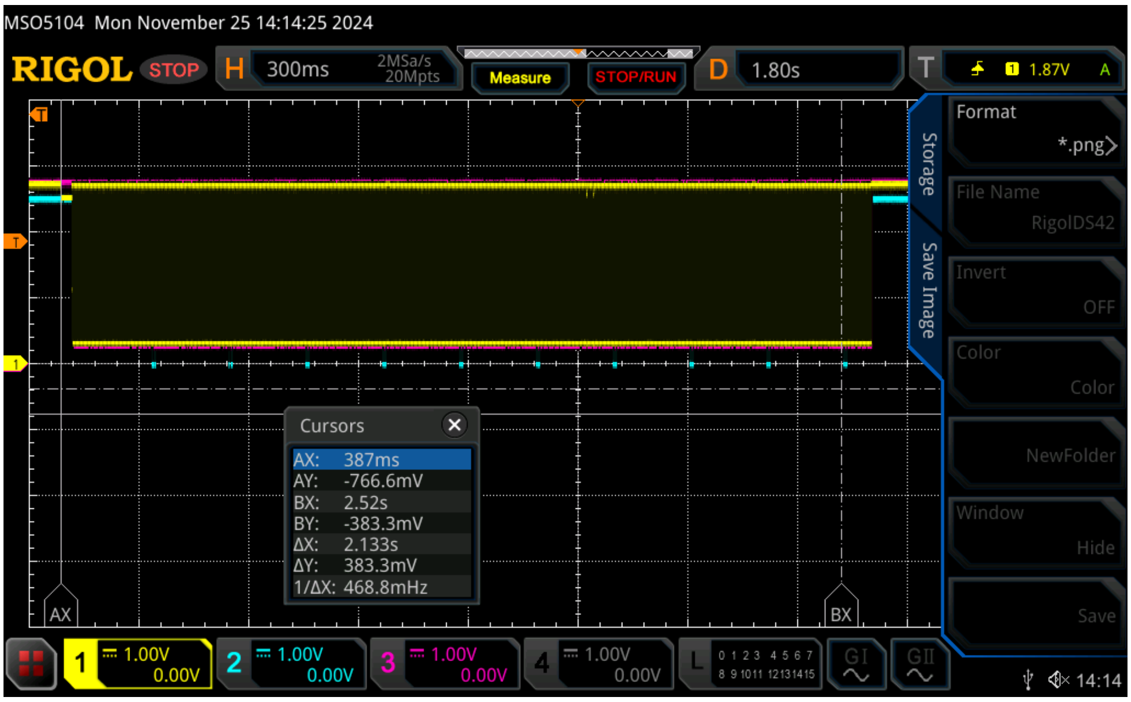
Figure 4.1: Graph that show how SPI Implementation on BLE Client impacts.

For the first Frequency Response Packets values, both graphs assume similar values. From value equal to 40 to up, the graphs assume different behavior: that one without SPI Implementation on BLE Client continues to go up until it reaches a throughput of more than 1 Mbps at Frequency Response Packets equal to 200 and finds a plateau. On the other hand, the graph depicting the system's behavior with the definitive version, at Frequency Response Packets equal to 100, reaches the plateau.

The values of throughput were obtained calculating the throughput from the oscilloscope captures with the equation below:

$$\text{Throughput} = \frac{8\,\text{bits} \times 244\,\text{bytes}/\#\text{packet} \times \#\text{packets}}{\text{time}} \tag{4.1}$$

In Fig. 4.2 is shown the way to obtain, from the oscilloscope capture, the value of Server throughput influenced by responses.



```
Cursors
AX:    387ms
AY:    -766.6mV
BX:    2.52s
BY:    -383.3mV
ΔX:    2.133s
ΔY:    383.3mV
1/ΔX:  468.8mHz
```

Figure 4.2: Oscilloscope capture of the Server throughput influenced by responses measurement. 1000 packets are sent and a Frequency Response Packet equal to 100 is set. In yellow the SPI transfer signal from the External module, the BLE Server transfer signal in purple and the responses received from the Client signal in light blue.

These signals, as next ones, are obtained exploiting the GPIO pins of each boards and configuring them in order to invert the state with the *bsp_board_led_invert()* function.

It is also possible to focus the attention on cursors AX and BX. The AX cursor is positioned in correspondence of the beginning of the SPI transfer meanwhile the BX cursor is positioned where the first packet transfer of the last response occurs. The motive of the BX cursor positioning is due to the fact that when the last BLE packet is sent, then the first packet of the last response is sent. In this way, the time indicated by the cursors is considered in the Eq. 5.1.

The last packets, in addition to the Client responses, are 50 packets more than the 1000 packets set to be sent, in order to ensure that all 1000 packets are transferred from the Server to the Client.

4.3 SPI Implementation also on BLE Client

This paragraph shows the results obtained with the definitive version of the system, where all the boards are used. In Fig. 4.3 is shown the final setup.

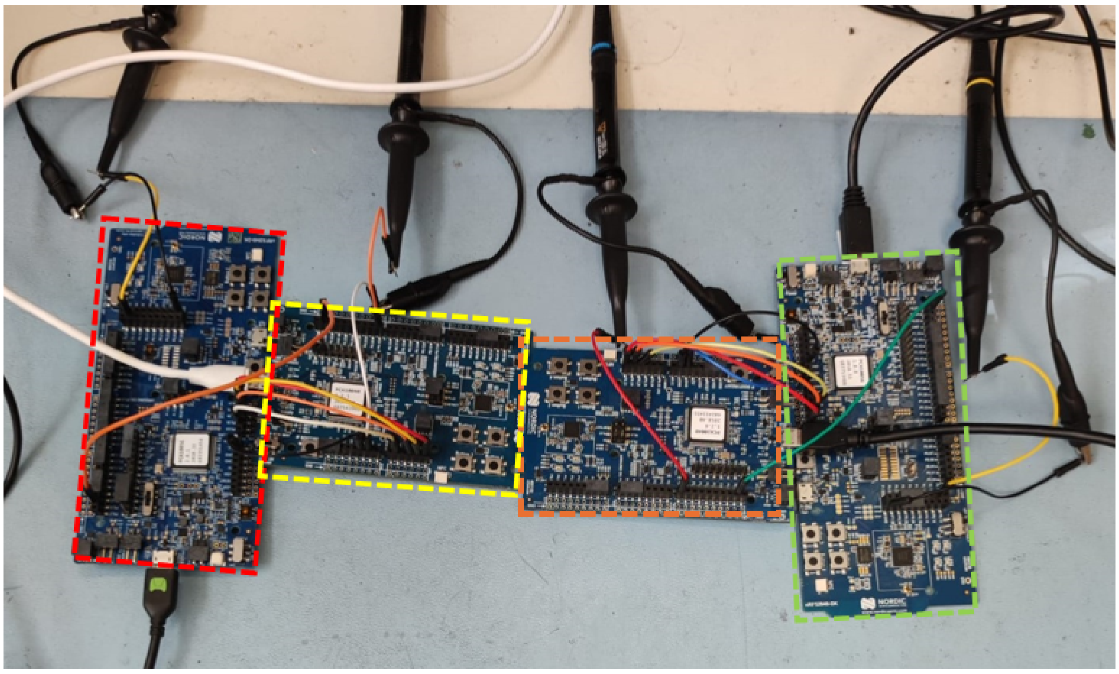


Figure 4.3: Final setup of the boards. Highlited in red the SPI Master External, in yellow the BLE External module, in orange the BLE Implantable Module and in green the SPI Master Implantable.

In Fig. 4.3, it is also possible to see the probes connected to the board bringing signal to the oscilloscope. This setup is the same saw in Fig. 2.2 with a different disposition on the table: the BLE modules facing each one with the other to ensure the antennas proximity.

To understand how the BLE behaves in the presence of biological tissue, the setup in Fig. 4.3 was modified by putting a chicken slice between the nRF52832 boards that play the role of BLE Client and BLE Server. It was added so that the biological tissue was interposed between the antennas on the boards.

The Fig. 4.4 shows the setup that was implemented to conduct the test.

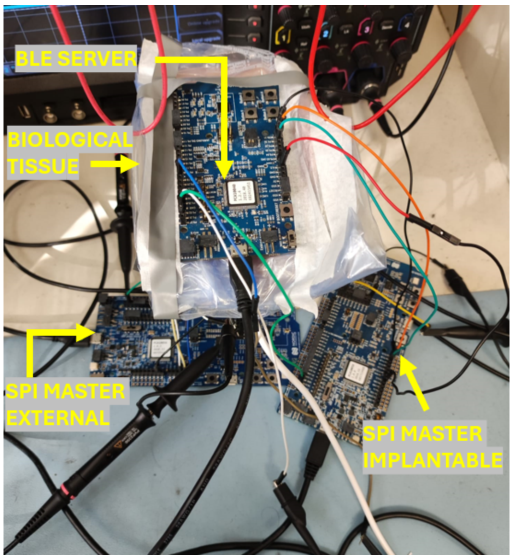


Figure 4.4: Setup with biological tissue from the top.

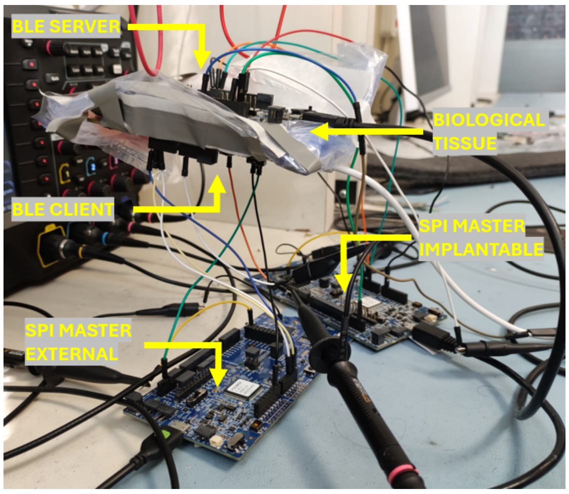


Figure 4.5: Setup with biological tissue from the side. From this perspective is possible to see the BLE Client board positionated under the biological tissue.

As observed, the BLE boards and the chicken are kept together via dedicated support. The BLE boards suspended, remain linked via SPI to the boards placed on the table, which act as SPI Masters.

The results obtained with these configurations are shown in Fig. 4.6, Fig. 4.7, Fig. 4.9. As with the Tab. 4.1 and Tab. 4.2, the values in the graphs represent the median of the obtained results, along with an error band encompassing the maximum and minimum values.

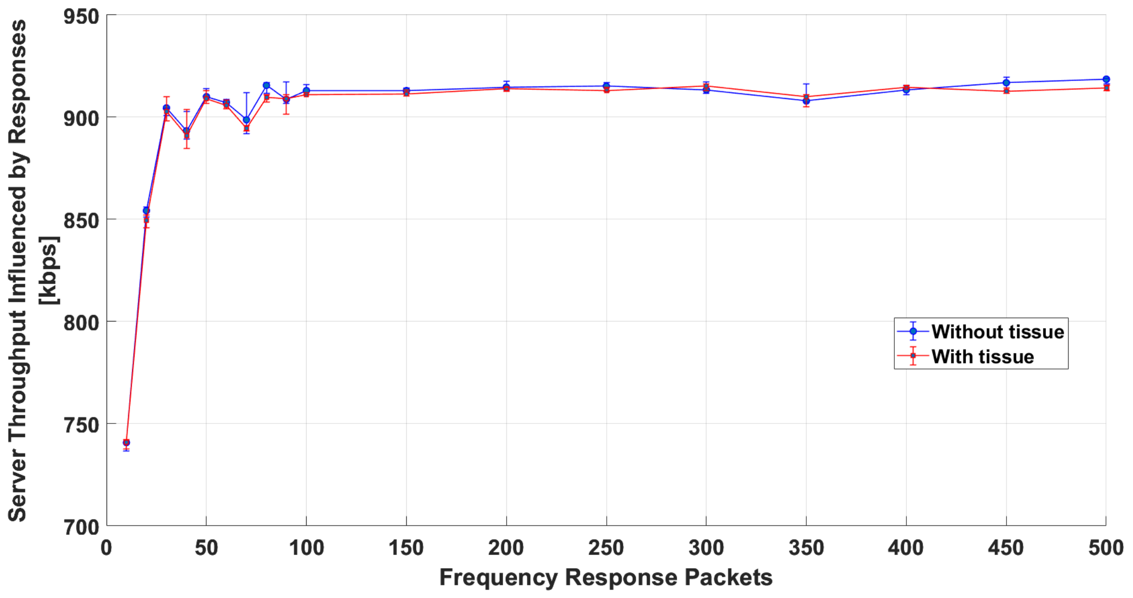


Figure 4.6: Server throughput influenced by and without tissue.

The values shown in Fig. 4.6 are obtained as depicted in Fig. 4.2.

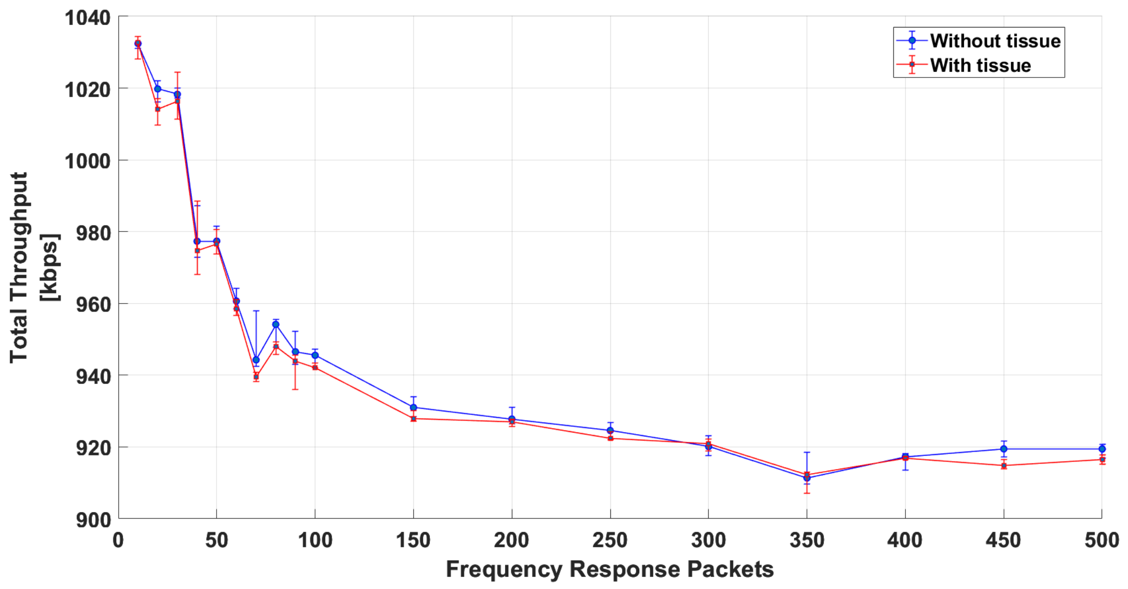
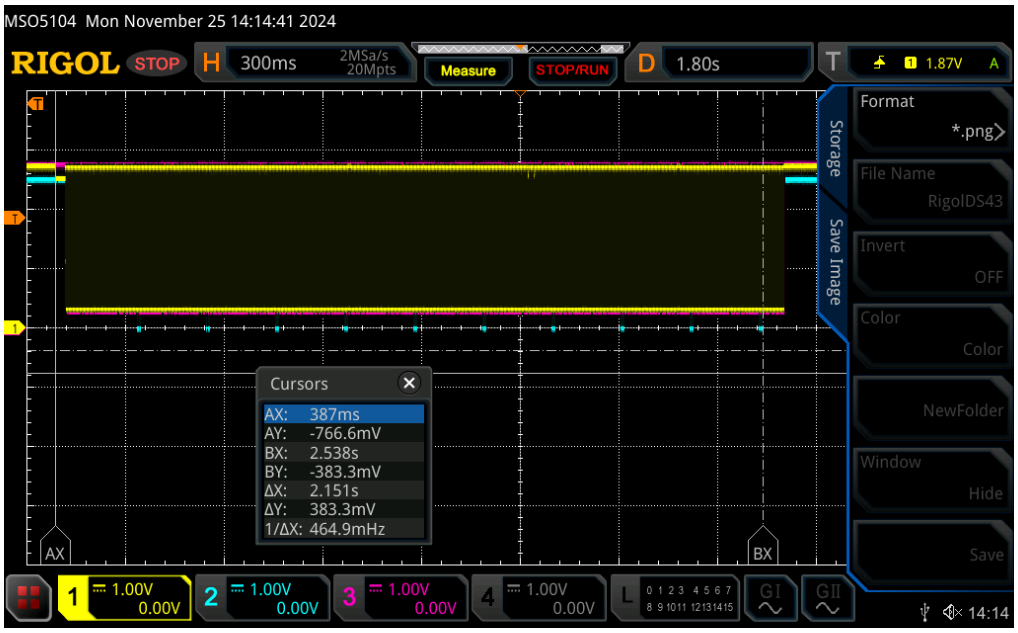


Figure 4.7: Total throughput with and without tissue.

The graph in Fig. 4.7 is obtained taking values from the oscilloscope as in Fig. 4.8.



Cursors

AX:	387ms
AY:	-766.6mV
BX:	2.538s
BY:	-383.3mV
ΔX:	2.151s
ΔY:	383.3mV
1/ΔX:	464.9mHz

Figure 4.8: Oscilloscope capture of the total throughput measurement. 1000 packets are sent and a Frequency Response Packet equal to 100 is set. In yellow the SPI transfer signal from the External module, the BLE Server transfer signal in purple and the responses received from the Client signal in light blue.

The focus on cursors allows to understand that the AX cursor is set at the beginning of the SPI transfer signal from the Implantable module, but opposite to Fig. 4.2, the BX cursor is located on the last packet of the last response.

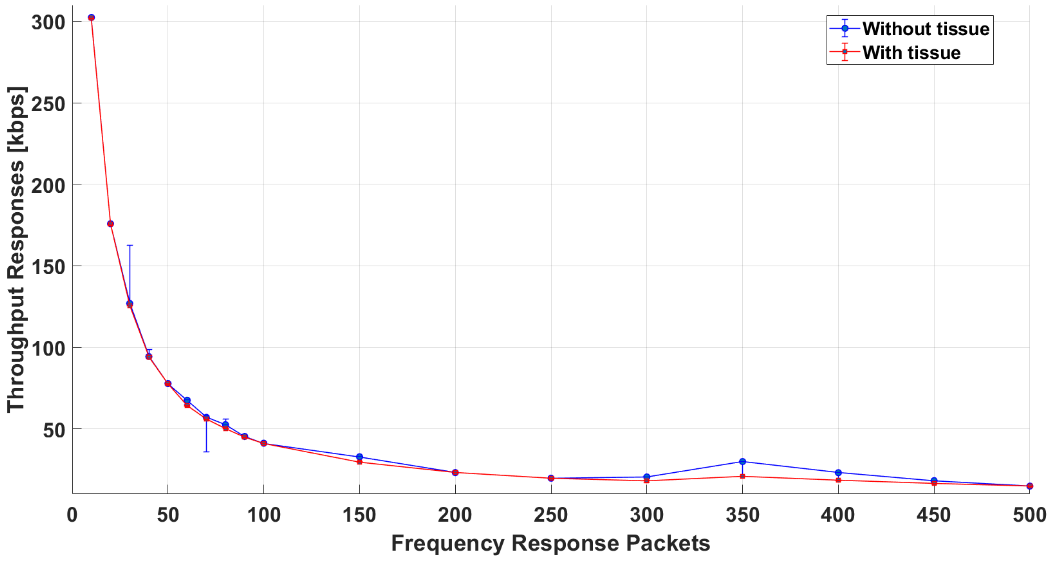
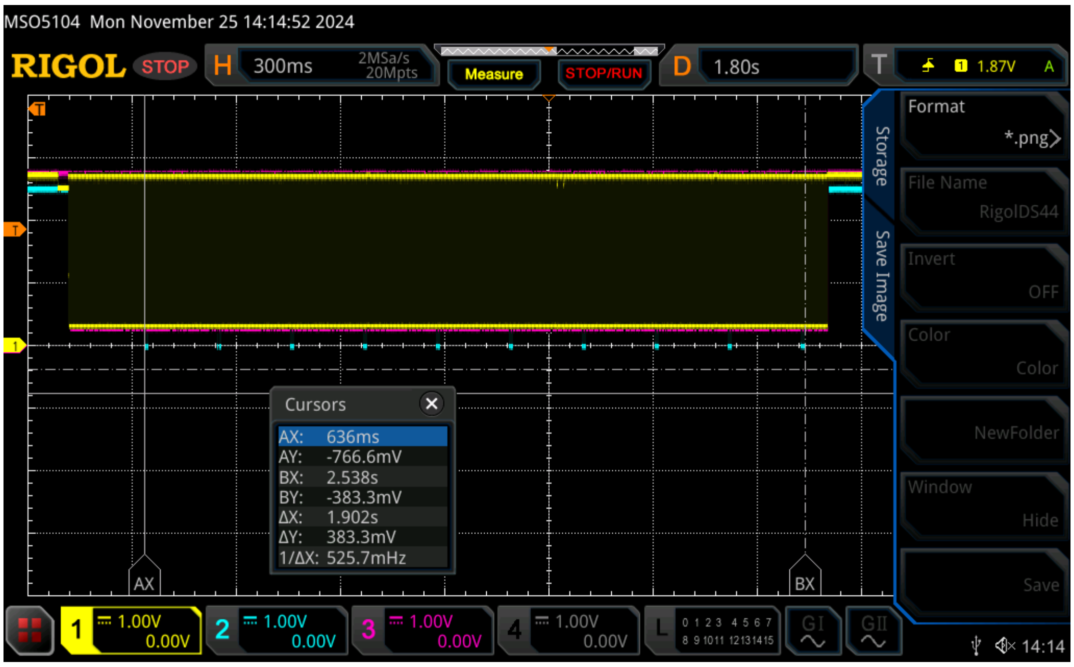


Figure 4.9: Throughput of the responses with and without tissue.

Finally, the values in the graph in Fig. 4.9 are obtained by setting the cursors of the oscilloscope as in Fig. 4.10: the AX cursor on the first packet of the first response and the BX cursor on the last packet of the last response.



```
Cursors
AX:    636ms
AY:    -766.6mV
BX:    2.538s
BY:    -383.3mV
ΔX:    1.902s
ΔY:    383.3mV
1/ΔX: 525.7mHz
```

Figure 4.10: Oscilloscope capture of the response throughput measurement. 1000 packets are sent and a Frequency Response Packet equal to 100 is set.

4.4 Discussion of the results

It is possible to catch the significant impact of the SPI protocol on the data rate of the BLE Server by noticing the variation of throughput values from Tab. 4.2 to Fig. 4.1 to the blue line of Fig. 4.6. By sending 1000 packets from the Server to the Client, the values pass from about 1.3 Mbps to 1.02 Mbps, resulting in a decrease of 21.5% . Implementing the SPI protocol on the External module, the values pass from 1.02 Mbps to about 919 kbps (also in this case are considered the two responses), leading to a decrease of about 10%.

From the graphs in Fig. 4.6, Fig. 4.7 and Fig. 4.9 it is possible to understand that there is no appreciable difference between the setup with biological tissue interposed between the BLE modules, as shown in Fig. 4.4 and Fig. 4.5, and the setup without biological tissue, as depicted in Fig. 4.3.

Observing the graph with SPI implementation on BLE Client in Fig. 4.1 (which is equivalent to the without-tissue line on the graph in Fig. 4.6) is possible to understand that from the Frequency Response Packet equal to 40 the values of Server throughput influenced by Client response are pretty similar, guaranteeing values from a minimum value of 893.36 kbps to a maximum of 918.59 kbps without biological tissue and values from a minimum of 890.92 kbps to a maximum of 914.27 kbps with biological tissue. By setting Frequency Response Packets of 15, is possible to meet the specification of the project of 800 kbps, as presented in the Abstract and explained in the Requirements section, both for without-tissue line and the with-tissue line. By sending less frequently response packets, the Server throughput influenced by Client responses increases.

It is possible to visualize in Fig. 4.7 that from Frequency Response Packets equal to 150, the Total throughput ranges from a maximum of 930.99 kbps to a minimum of 919.45 kbps without biological tissue and from a maximum of 927.97 kbps to a minimum of 916.45 kbps. It is also an interesting notice that the maximum Total throughput is 1032 kbps and is obtained by selecting a Frequency Responses Packets of 35, both for the without-tissue line and the with-tissue line.

On the other hand, taking into account Throughput response requirements, it is possible to individuate on Fig. 4.9 that the values on the graph are pretty similar from Frequency Responses Packets equal to 150: from a maximum value of 32.84 kbps to 14.80 kbps for the without-tissue case and from 29.14 kbps to 14.80 kbps. To meet the requirement of 32 kbps data rate for Client-to-Server packets, indicated in Abstract and explained in the Requirements section, the upper-Frequency Response Packets limit is 150 for without-tissue configuration and about 140 for the with-tissue case.

Chapter 5

Conclusion

This project allows to send data via BLE between the Implantable Module and the External Module. Data generated from SPI Master Implantable come via SPI to the BLE Implantable module, then are sent via BLE to the BLE External module. Meanwhile, thanks to handshake logic based on GPIO, the BLE External module sends response packets created by SPI Master External at a set frequency. In Fig. 5.1, the flowchart of the global system.

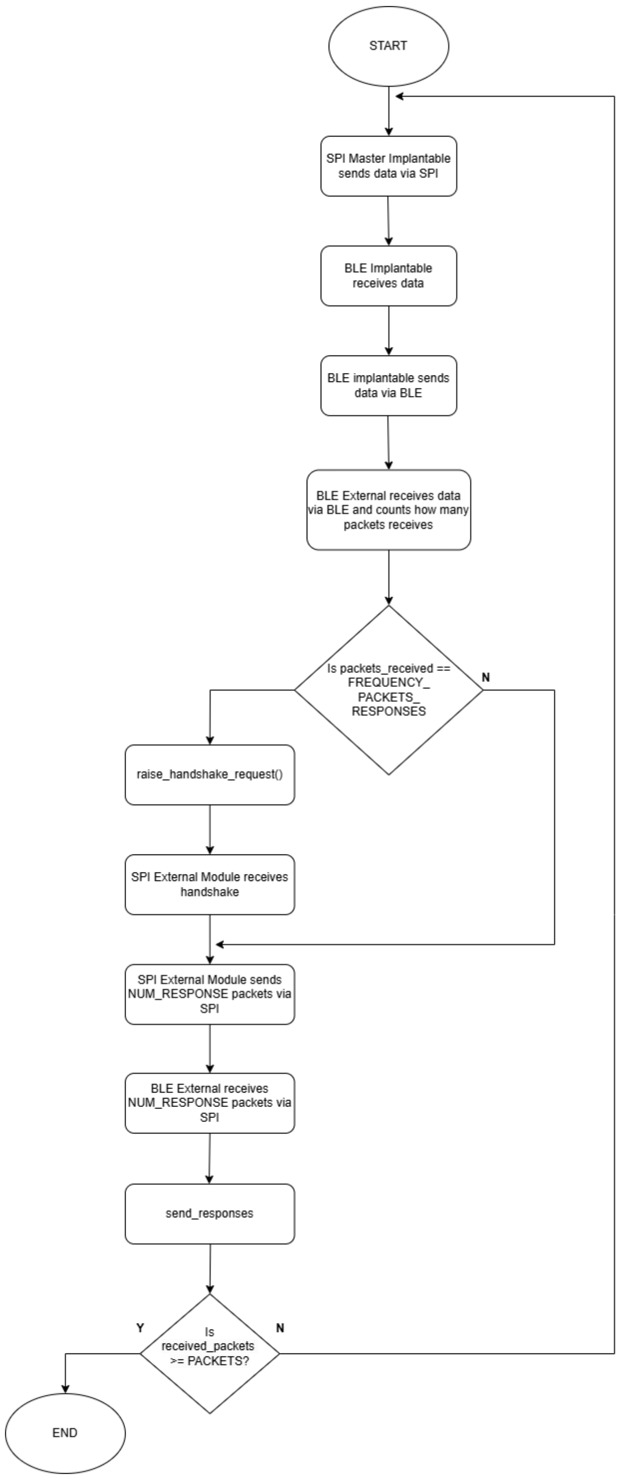

START
SPI Master Implantable sends data via SPI
BLE Implantable receives data
BLE implantable sends data via BLE
BLE External receives data via BLE and counts how many packets receives
Is packets_received == FREQUENCY_ PACKETS_ RESPONSES
N
raise_handshake_request()
SPI External Module receives handshake
SPI External Module sends NUM_RESPONSE packets via SPI
BLE External receives NUM_RESPONSE packets via SPI
send_responses
Y
Is received_packets >= PACKETS?
N
END

Figure 5.1: General flowchart that explains the data flux between the boards.

The system was tested with two different configurations: with biological tissue and without biological tissue. No significant variations are appreciable between these two configurations.

In each configuration, the tests guarantee the project specifications in each direction. Since the presence of responses from the External Module slows down the Server throughput performance, it is necessary to individuate the Frequency Response Packets limit, which allows it to meet the specifications. Indeed, the 800 kbps throughput required from the Server is possible to obtain with Frequency Response Packets set at 15 for the case with the tissue presence and without tissue as well. By increasing the Frequency Response Packets value, the Server throughput influenced by Client responses increases, since responses by Client are sent more rarely.

On the other hand, the 32 kbps throughput required from the Client is reached for Frequency Response Packets set at 150 for the without-tissue case and at 140 for the with-tissue case.

Knowing that, to meet simultaneously the specifications required in each direction, in the:

- without-tissue case, it needs to choose a Frequency Response Packets equal to 150. In this way, the total throughput results as 930.99 kbps
- with-tissue case, it needs to choose a Frequency Response Packets equal to 140. In this way the total throughput results as 927.97 kbps.

Crucial consideration must be made on system performance based on CI and package size settings. The trade-off is the lowest CI and the maximum packet dimension.

The lowest CI duration reduces latency, which is ideal for real-time applications like this project. From what was seen in Tab.1 and Tab.2, increasing the number of packages from 100 to 1000 would increase the CI. By introducing latencies with the implementation of the SPI protocol in the two modules, it was necessary to reduce the duration of the CI.

On the other hand, the maximum packet dimension setting made possible the reduction of CEs number and, since each CE requires overheads, in this way, the throughput is enhanced. This is made possible thanks to the DLE, as described on *main.c*.

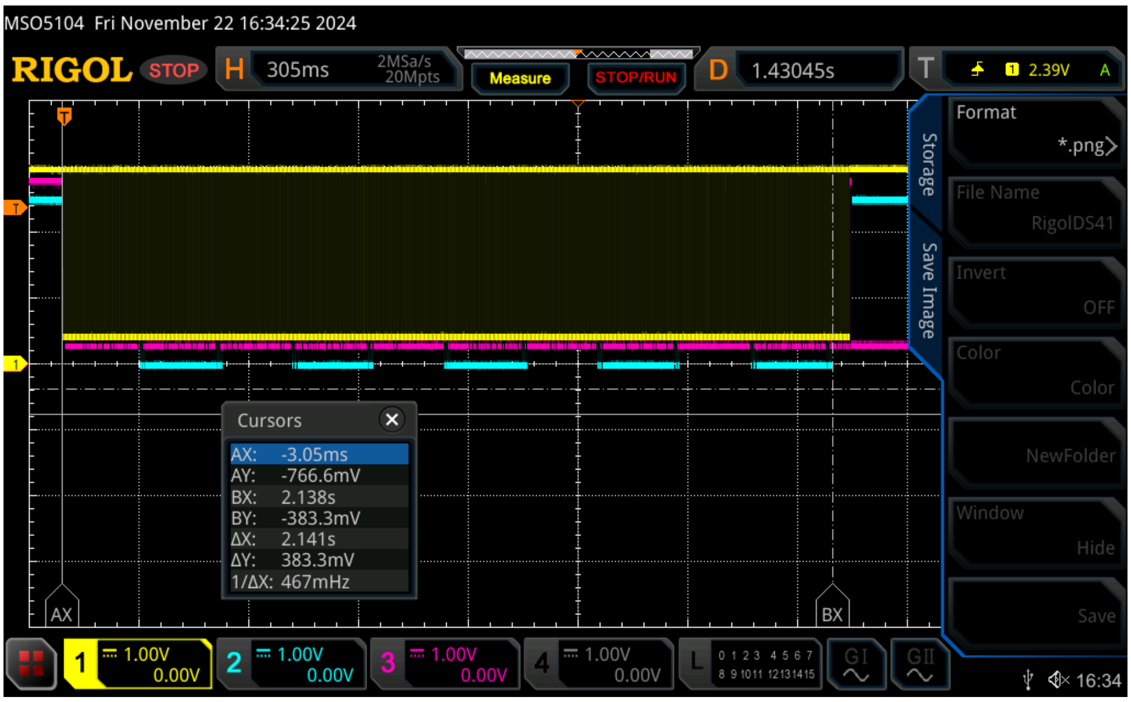
5.1 Limitations

A limitation that the project has incurred is the NRF_ERROR_RESOURCE. This error is present because, including responses from the Client, too many notifications are queued. This condition could be possible based on the CI interval decision.

It is true that the lowest is the CI the highest is the throughput because it is possible to do multiple connection intervals with data transmitted in a short amount of time (this is the motive it was decided at first, in order to maximize the Server throughput), but it is not enough to handle both all the Server packets and all the Client packets.

One cause could be the fact that "the link between a peripheral and a central can be asymmetric, meaning that the packets from the peripheral can be sent using the 1M PHY, while packets from the central can be sent using the 2M PHY" [1]. This asymmetric situation is observed in the tests, although what is coded and shown in Development and Testing of Codes Implementations section, setting parameters.

Since the motivation for this error could also be the setting of the CI, it was necessary to add a delay. Different tests have been conducted to understand which value of delay overcomes the problem, and a delay of 800 µs overcomes the problem just in case NUM_RESPONSES is set at 5. However, any value of delay overcomes the NRF_ERROR_RESOURCES over five response packets. The positive part is that the delay of 800 µs has not worsened the throughput which results 957 kbps setting a FREQUENCY_RESPONSE_PACKETS of 100, as the Fig. 5.2 shown.


MSO5104 Fri November 22 16:34:25 2024

H 305ms 2MSa/s 20Mpts
D 1.43045s
T 2.39V

Cursors
AX: -3.05ms
AY: -766.6mV
BX: 2.138s
BY: -383.3mV
ΔX: 2.141s
ΔY: 383.3mV
1/ΔX: 467mHz

Format *.png
File Name RigolDS41
Invert OFF
Color Color
Window Hide

1 1.00V 0.00V
2 1.00V 0.00V
3 1.00V 0.00V
4 1.00V 0.00V

Figure 5.2: Oscilloscope capture with 5-packets response of 244 bytes each one.

Another limitation was the setup used to conduct the tests in the presence of biological tissue, as shown in Fig. 4.4 and Fig. 4.5. It does not allow the stabilization of the BLE Implantable board, BLE External board or the biological tissue between the two boards. The difficulties with interposition between the boards and the slice of tissue is because the board positioned under the tissue can't be placed on the table since it faces the pins connected to the SPI Master to the table.

5.2 Possible enhanchements

This subsection takes room for possible enhancement of the issue in the Limitations subsection.

About the NRF_ERROR_RESOURCES, the problem could be overcome by splitting the code shared by the BLE Implantable module and the BLE External module. In this way, it is possible to choose two different CIs, one for the Server and another for the Client. This solution could also overcome the condition of asymmetric PHYs between peripheral and central since, in this project, the TX/RX transmission of each BLE module is managed by only one code, *ble_app_att_mtu_throughput*, as explained with in Development and Testing of Codes Implementations. The solution to this problem could also be to omit the Nordic DK and use a custom system.

On the other hand, the setup with the biological tissue can be realized with a more stable system, which allows for safer positioning and ensures the presence of the tissue between the BLE Implantable and BLE External antennas without constant control by the operator.

Another future improvement could be the realization of a test closer to the final application, where the Implantable module is completely surrounded by biological tissue as it is surgically inserted into the body, close to the amputation like in Fig. 1.15. For this purpose, as in Fig. 4.4 and Fig. 4.5, the BLE Implantable module antenna is not completely isolated from the BLE External module antenna, but only separated by a slice of biological tissue, while around the BLE External module antenna there is air and no other biological obstacles.

Finally, it is important to note that in the final configuration of the project, the data will come

- to the BLE Client via mechatronic sensors, converted in analog signal by a 9–bit DAC and sampled at 10 kHz frequency, with a 10% duty cycle, on two different 16–bit channels;
- to the BLE Server from the nervous system, captured by bidirectional electrodes, converted by a 13–bit ADC and sampled with at 10 kHz frequency on five different 16–bit channels.

Differently, in this thesis, dummy data was simulated with SPI Master External and SPI Master Implantable. Then in the final configuration, in addition to ensuring data flow, the other objective is to maintain throughput performance to prevent it from being affected by future necessary developments.

Bibliography

[1] M. Afaneh, *Intro to Bluetooth Low Energy*. Novel Bits, LLC, 2018. https://www.novelbits.io.

[2] ProgrammerSought, "Bluetooth low energy ble connection events, connection parameters and update methods (program interpretation)," 2024. https://www.programmersought.com/article/26076936373/.

[3] R. Heydon, *Bluetooth Low Energy: The Developer's Handbook*. Upper Saddle River, NJ: Pearson Education, 2012.

[4] Punch Through, "Maximizing ble throughput part 3: Data length extension (dle)," 2016. https://punchthrough.com/maximizing-ble-throughput-part-3-data-length-extension-dle-2/.

[5] École Polytechnique Fédérale de Lausanne (EPFL), "Amputees feel warmth in their missing hand," 2023. https://actu.epfl.ch/news/amputees-feel-warmth-in-their-missing-hand-2/.

[6] P. A. Starr, "Totally implantable bidirectional neural prostheses: A flexible platform for innovation in neuromodulation," *Frontiers in Neuroscience*, 2018. https://www.frontiersin.org/articles/10.3389/fnins.2018.00619/full.

[7] S. Raspopovic *et al.*, "Restoring natural sensory feedback in real-time bidirectional hand prostheses," *Science Translational Medicine*, vol. 6, no. 222, p. 222ra19, 2014.

[8] H. Kim *et al.*, "Advances in wireless, batteryless, implantable electronics for real-time, continuous physiological monitoring," *Nano-Micro Letters*, vol. 12, no. 1, pp. 1–20, 2020.

[9] A. Name *et al.*, "Senseback - an implantable system for bidirectional neural interfacing," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 18, no. 1, pp. 123–134, 2024.

[10] NerveRepack Project, "About nerverepack," 2024. https://www.nerverepack.eu/about-7.

[11] F. I. for Reliability and M. IZM, "Nerverepack project," *Fraunhofer IZM Tech News*, 2024. https://www.izm.fraunhofer.de/en/news_events/tech_news/nerverepack.html.

[12] Wireshark, "Wireshark - downloads." https://www.wireshark.org/

download.html.

[13] nRF52840 Dongle, "nrf52840 dongle - development." https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle.

[14] nRF Connect for Mobile, "nrf connect for mobile - downloads." https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-mobile.

[15] SPIDriver, "Spidriver - downloads." https://spidriver.com/.

[16] Biology LibreTexts, "16.5: The action potential," *Biology LibreTexts*, 2024. Disponibile su: https://bio.libretexts.org/Courses/Lumen_Learning/Anatomy_and_Physiology_I_(Lumen)/16%3A_Module_14-_The_Nervous_System_and_Nervous_Tissue/16.05%3A_The_Action_Potential.

[17] Nordic Semiconductor, "nrf52832 product brief." https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52832.

[18] Nordic Semiconductor, "nrf52840 product brief." https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK,Accessed:2024-11-02.

[19] Nordic Semiconductor, "nrf52 dk development kit." https://www.nordicsemi.com/Products/Development-hardware/nRF52-DK.

[20] Nordic Semiconductor, "nrf52840 development kit." https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK.

[21] SEGGER, "Segger embedded studio - downloads." https://www.segger.com/downloads/embedded-studio/.

[22] nRF Connect for Desktop, "nrf connect for desktop - downloads." https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop.

[23] N. S. D. Community, "Can i increase nrf_sdh_ble_gatt_max_mtu_size over 247?." https://devzone.nordicsemi.com/f/nordic-q-a/94282/can-i-increase-nrf_sdh_ble_gatt_max_mtu_size-over-247.

[24] Bluetooth SIG, "Core specification 4.2," 2024. https://www.bluetooth.com/specifications/specs/core-specification-4-2/.