BLE-BASED MOTION DATA LOGGER

Documentation

Professors: Cristiana Bolchini, Antonio Rosario Miele

Alessandro Albrigo
Matricola 225759 – Cod. Pers. 10889223

Giuseppe Boniver Conte
Matricola 212975 – Cod. Pers. 10815020

POLITECNICO
MILANO 1863

August 2025

Contents

1 Peripheral Part

2 Introduction

The aim of this project was to re-engineer the firmware and communication system of an IoT device. The original device was based on a microcontroller interfacing with an accelerometer and transmitting data to a base station through the ANT protocol.
The system included:

- A firmware written in `C` running on the microcontroller.
- A Python-based client running on the base station.

While functional, the original implementation posed limitations in terms of **compatibility**, **energy efficiency**, and **extensibility**, particularly due to the use of the ANT protocol, which is less widely supported compared to modern alternatives.

2.1 Purpose of the project

The main objective of this project was to replace the ANT communication protocol with **Bluetooth Low Energy (BLE)**, in order to:

- Improve compatibility with modern devices (e.g., smartphones, laptops).
- Simplify the connection setup process.
- Take advantage of BLE's native support in many platforms.

To achieve this, the project involved the following key activities:

1. Redesigning the firmware to use BLE for data transmission.
2. Implementing a GATT server on the microcontroller.
3. Defining BLE services and characteristics for timestamping, configuration, data acquisition and battery monitoring.
4. Logging raw data to a microSD card for offline analysis.
5. Maintaining compatibility with the device's existing hardware, including the IMU and power management system.

3 System Overview

The redesigned system is structured around a microcontroller-based board equipped with an inertial measurement unit (IMU), BLE communication capabilities, and an SD card for local data storage. The device operates in coordination with a BLE central (e.g., mobile application or PC client) and follows a well-defined interaction protocol.
At a high level, the device progresses through a series of operational phases orchestrated by a Finite State Machine (FSM), which determines the board's behavior based on the BLE commands received and internal conditions.
The system's operational flow is as follows:

1. **Power-on and Advertising:** Upon startup, the device advertises itself over BLE and waits for a connection from a central.
2. **Synchronization:** Once connected, the client synchronizes the device's internal clock by sending a timestamp.
3. **Configuration:** The client provides sampling parameters such as acquisition frequency and initial counter settings.
4. **Acquisition:** The board collects data from the IMU and transmits it via BLE while simultaneously logging it to the SD card.
5. **Session Termination:** The client can command the board to stop acquisition, enter sleep mode, or reset for a new session.

This behavior is encapsulated in a state-driven architecture, where each state (e.g., `WAIT_TIMESTAMP`, `WAIT_CONFIGURATION`, `ACQUISITION`, etc.) corresponds to a well-defined step in the BLE protocol. Transitions between states are triggered by BLE commands or timeouts.
This architecture enhances modularity, robustness to disconnections, and clarity of execution flow.

4 Technical Specifications

The system features the following specifications:

- Microcontroller: Nordic nRF52840 (Arduino Nano 33 BLE).
- Communication protocol: Bluetooth Low Energy (BLE) 5.0.
- Motion sensing: 9-axis IMU (accelerometer, gyroscope, magnetometer).
- Sampling frequency: configurable via BLE.
- Storage: microSD card via SPI interface.
- Power source: 3.7V Li-Po rechargeable battery.
- Data format: raw accelerometer data, sent over BLE and logged in hexadecimal format.

5 Hardware Used

The system is built around the Arduino Nano 33 BLE board, featuring a Nordic nRF52840 microcontroller. The main hardware components used in the project are:

Arduino Nano 33 BLE

- Microcontroller: Nordic nRF52840 (32-bit ARM Cortex-M4 @ 64 MHz)
- Memory: 1 MB Flash, 256 KB RAM
- Wireless: Bluetooth® 5.0 Low Energy
- IMU: LSM9DS1 (9-axis: accelerometer, gyroscope, magnetometer)
- Operating voltage: 3.3 V (logic level)
- Micro USB interface for power and programming

SD Card Module

- SPI interface (MOSI, MISO, SCK, CS, VDD, GND).
- Used for local logging of raw IMU data in hexadecimal format.
- Interfaced via the `SPI.h` and `SD.h` libraries.

Power Supply

- 3.7 V Li-Po battery.
- Monitored via a voltage divider connected to an analog input (A0).
- Battery level estimation and BLE notification implemented in firmware.

6 Main Features

The system provides several key functionalities aimed at efficient motion data acquisition and BLE communication. The main features are:

- **BLE-Based Configuration:** the device receives configuration parameters (e.g., sampling frequency, initial counter value) via BLE characteristics before data acquisition begins.
- **Timestamp Synchronization:** a global timestamp is sent to the device to synchronize time-based logging, ensuring coherence across multiple devices.
- **IMU Sampling and Formatting:** the device samples the built-in 3-axis accelerometer (LSM9DS1), reads raw values and packs them into a fixed-size data structure.
- **Data Transmission via BLE Notifications:** sampled data is transmitted to a central BLE device (e.g., PC or smartphone) using BLE notifications.
- **Local Storage on SD Card:** in parallel with BLE transmission, data is also stored locally on a microSD card in raw hexadecimal format.
- **Battery Level Monitoring:** the battery charge level is periodically read and notified to the central device using a BLE battery service.
- **State Machine Based Operation:** the firmware is structured around a finite state machine (FSM) that manages the connection and communication phases, sampling process, and disconnection recovery logic.
- **Error Handling and Sleep Mode:** Upon unexpected disconnection, the device attempts to reconnect for a fixed timeout period. If the central device fails to reconnect within this time, the system enters a dedicated "Final" state. This mechanism prevents uncontrolled data acquisition and ensures safe termination of the session.

7 Software Overview

The software running on the microcontroller is written in C using the Arduino framework. It is structured into several functional modules that manage communication, data acquisition, and storage. The system is controlled by a finite state machine (FSM) that ensures proper transitions across different operating phases.

7.1 Main Modules

- **BLE Manager:** Handles Bluetooth Low Energy communication with the central device. It receives commands (timestamp, configuration, start/stop) and transmits sampled data.
- **IMU Acquisition:** Periodically reads raw acceleration values from the onboard sensor.
- **Data Logger:** Stores acquired data both in text and hexadecimal format onto an SD card.
- **Battery Monitor:** Reads the battery level via ADC and periodically notifies the central device.
- **State Machine:** Controls the workflow of the device, including connection phases, active acquisition, and Final state.

7.2 System Workflow

At power-on, the device waits to receive a timestamp and configuration parameters. Once an acknowledgment is sent, the system enters acquisition mode upon receiving a start command. Data is sampled and transmitted until a stop signal or unexpected disconnection occurs. In case of prolonged disconnection, the device enters a dedicated Final state to avoid inconsistent behavior.

7.3 External Libraries Used

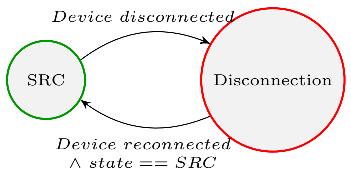
The firmware relies on the following external libraries to manage hardware and system functionalities:

- **ArduinoBLE.h** – Provides support for Bluetooth Low Energy communication, used to send and receive data and commands.
- **Arduino_LSM9DS1_AAAC.h** – Handles the interface with the onboard LSM9DS1 inertial measurement unit (IMU) for reading acceleration data.
- **Arduino_JSON.h** – Used to parse the JSON configuration file stored on the SD card during initialization.
- **SPI.h** – Required for communication with the SD card using the SPI protocol.
- **SD.h** – Provides functions for reading and writing files on the SD card.
- **TimeLib.h** – Used for managing and formatting timestamps received from the central device.

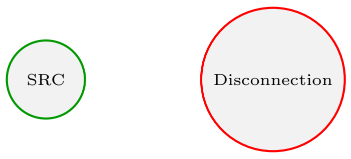
8 Finite State Machine (FSM)

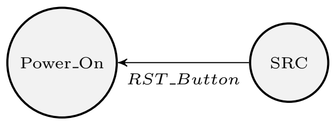
The system operates based on a finite state machine (FSM) that defines the various states and transitions during its operation. The FSM is designed to handle the different phases of the device's lifecycle, including initialization, configuration, data acquisition, and disconnection handling.

8.1 FSM

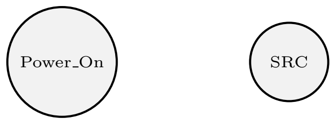


equals to:





equals to:



This behavior is due to the fact that, regardless of the current state, pressing the reset button always returns the device to the initial state, effectively restarting the board.

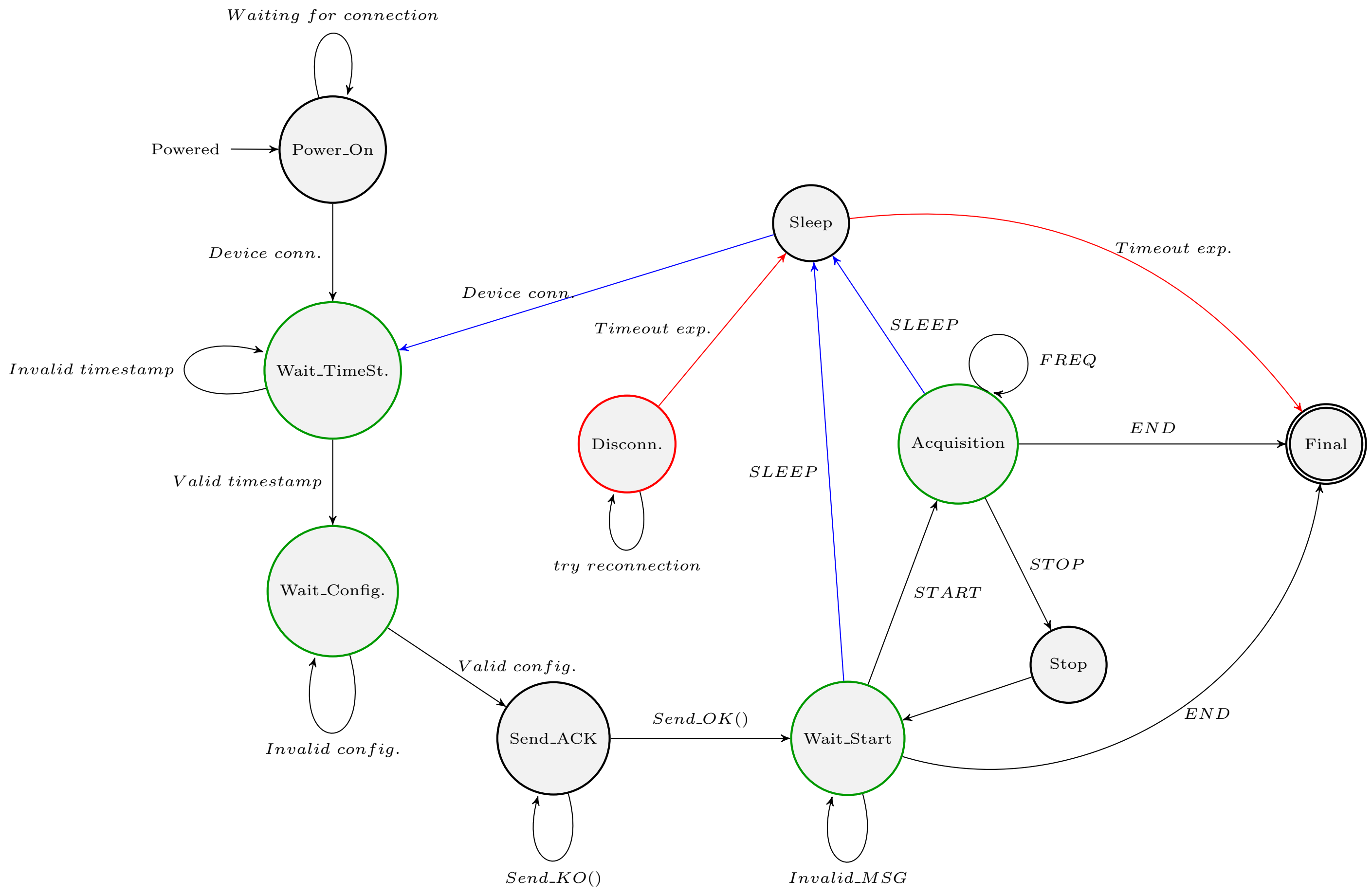


Figure 1: FSM Simplified

8.2 FSM Description

The system is governed by a Finite State Machine (FSM) that ensures consistent behavior throughout its operational lifecycle. The FSM defines a set of discrete states and transition rules that the device follows depending on user input, configuration data, and connection status. To improve readability and modular understanding, the FSM is visually simplified into two subgraphs:

- The **SRC/Disconnection loop** (top of the figure) abstracts the behavior related to the device's connection management before the acquisition process begins.
- The **main FSM diagram** (bottom) outlines the complete lifecycle, including startup, configuration, data acquisition, disconnection handling, and critical errors.

8.3 Transition colors:

The simplification is introduced solely for the sake of clarity. In this representation, some transitions between states are intentionally omitted to ensure a more concise and readable FSM.

Blue arrows indicate transitions occurring when a computation is deliberately terminated to start a new one.

Red arrows indicate transitions triggered when the reconnection timeout to the BLE central is exceeded.

8.4 State Descriptions

- **State_Power_On:** Initial state entered at boot. The device is set up and waits for a BLE connection.
- **State_Wait_TimeStamp:** Once a BLE central device is connected, it waits for a valid timestamp from the client. Invalid timestamps cause a loop.
- **State_Wait_Configuration:** Waits for the configuration message. If the configuration is invalid, the device remains in this state.
- **State_Send_ACK:** Sends acknowledgment (`ACK_OK` or `ACK_KO`) to the client after verifying configuration.
- **State_Wait_Start:** Waits for the `START_CAMPAIGN` message before starting data acquisition.
- **State_Acquisition:** Actively samples IMU data and logs it to SD. This state can receive frequency updates (`FREQ`) or be interrupted by a stop command.
- **State_Disconnection:** State entered when the BLE connection is lost. The device tries to reconnect for a limited time.
- **State_Sleep:** Low-power mode entered if reconnection fails within the timeout threshold or when the client wants to start a new computation.
- **State_Stop:** Used to handle graceful interruption of a campaign.
- **State_Final:** Final state entered when timeout for disconnection is expired or when the computation end due to END command sent by the client.

9 Data Structures and Configurations

This section provides an overview of the internal data structures and configurable parameters used by the system. These elements define how the device operates, handles communication, and stores data during an acquisition campaign.

9.1 Data Structures

The firmware defines several custom data structures to manage the acquisition flow:

- `Datapacket_t`: a structure containing the sampling counter and the raw accelerometer data in 16-bit format. It is transmitted over BLE and stored in the SD card.
- `campaign_parameters_t`: a structure storing the acquisition parameters (e.g., sampling frequency and initial counter), which are loaded from a `config.json` file.

9.2 Configuration File

In the setup() function, is called setupInit() that reads a `config.json` file from the SD card. This file provides initial values for:

- **default_counter**: the initial counter value used to tag each sample;

- **default_freq**: the acquisition frequency, used to compute the sampling interval;
- **default_timer**: the timeout (in milliseconds) at which the battery voltage is sampled;
- **default_lost_connection_timeout**: the timeout (in milliseconds) for reconnect attempts before entering the Final state.

9.3 Runtime Configuration via BLE

The device can receive additional runtime configurations through BLE messages. These include:

- Timestamp synchronization (received before campaign start);
- Sampling frequency adjustments (sent during the campaign);
- Commands to start, stop, terminate the current computation and start new one, end the process or change acquisition parameters.

10 Software Structure and Control Flow

The software is organized around the classical Arduino `setup()` and `loop()` functions. The `setup()` routine performs one-time initializations such as BLE advertisement setup, IMU and SD initialization, and configuration loading from an external JSON file. The `loop()` function acts as the main execution cycle and is responsible for polling BLE events, acquiring data from the IMU when requested, handling disconnection timeouts, and managing state transitions defined in the FSM (Finite State Machine).
The behavior of the system is largely controlled by five main functions. Among them, three act as *callback handlers* for BLE characteristics. These are responsible for receiving commands from the BLE central and updating internal parameters and FSM states accordingly. Together, they serve as the logical controller of the board.

- `callbackCharacteristicTimestamp(...)`: This function is triggered upon receiving a BLE write on the timestamp characteristic. It decodes the timestamp sent from the central (encoded in 4-byte hexadecimal format), sets the internal system time using the `TimeLib.h` library, stores the timestamp string in the SD file, and transitions the FSM to `STATE_WAIT_CONFIGURATION`.
- `callbackCharacteristicConfiguration(...)`: This function handles different payload formats based on the number of received bytes. It supports three types of configuration messages:

  1. **3 bytes**: Initializes counter and sampling frequency.
  2. **2 bytes**: Updates sampling frequency dynamically.
  3. **1 byte**: Confirms initialization from the SD card JSON.

  If the configuration is valid and the timestamp was previously received, the FSM proceeds to the acknowledgment phase (`STATE_SEND_ACK`).
- `callbackCharacteristicStartStop(...)`: This function receives start and stop commands from the central. Based on the byte received, it transitions the FSM to one of the following states:

  – `STATE_ACQUISITION` for starting data collection.
  – `STATE_STOP` to pause acquisition.
  – `STATE_SLEEP` to end the current computation and start a new one opening a new data file.
  – `STATE_FINAL` to terminate the campaign and close the SD file.

  Any command received out of sequence or before proper initialization is safely ignored.
- `decideNextStateOnConnect(...)`: This function is invoked when the board is connected with the BLE central. It evaluates the current internal context (such as campaign status and previously exchanged messages) to determine the most appropriate state to transition to. The possible target states include:

  – `STATE_ACQUISITION`, if the campaign is ongoing or already started.
  – `STATE_WAIT_START`, if the configuration and acknowledgment are complete but acquisition has not yet begun.

  - `STATE_SEND_ACK`, if configuration is complete but the acknowledgment has not been sent yet.
  - `STATE_WAIT_CONFIGURATION` or `STATE_WAIT_TIMESTAMP`, depending on which initialization steps are still missing.

- `decideNextStateOnDisconnect(...)`: This function is called when a disconnection is detected, provided the system was not in the power-on state. It mirrors the logic of the previous function but operates under the assumption that a temporary loss of connection has occurred.

- `sendACK_OK(...)`: This auxiliary function is used to send a positive acknowledgment (ACK) message over BLE once a valid configuration has been received. It updates internal flags and advances the FSM to `STATE_WAIT_START`, preparing the system to receive the start command from the central device.

- The `loop()` function is responsible for real-time evaluation of connection status, BLE message polling, and execution of actions associated with the current state of the FSM. It also monitors timeouts during disconnection, manages reconnection attempts, and ensures state persistence. The following key tasks are handled:
  - Detect BLE central connection and trigger `decideNextStateOnConnect()`.
  - Handle disconnection and reconnection strategies with timeout support.
  - Execute sampling operations in `STATE_ACQUISITION`.
  - Manage shutdown and memory release in `STATE_SLEEP`.
  - Update battery level periodically if the device is connected.

11 BLE Protocol Design

Bluetooth Low Energy (BLE) is a wireless personal area network technology designed for short-range communication with low power consumption. In this project, BLE replaces the previous ANT protocol, enabling standardized interaction between the IoT device and a BLE central such as a smartphone, PC, or embedded gateway. The device is configured as a GATT (Generic Attribute Profile) server, which exposes multiple services and characteristics to the BLE central. Communication occurs via write and notify operations on these characteristics, each identified by a universally unique identifier (UUID).

12 BLE Communication Protocol

To synchronize and coordinate data collection between the BLE central and the IMU-based board, the protocol illustrated below is implemented.

12.1 Services and Characteristics

The main custom service is defined using a base UUID structure of the form:

Listing 1: Base UUID macro

```
#define AAAC_KIT_UUID(val) ("555a0002-" val "-467a-9538-01f0652c74e8")
#define UUID_SRV           AAAC_KIT_UUID("0000")
```

The following table summarizes the characteristics and services implemented:

Name	UUID	Access	Description
`ACK`	`555a0002-0010...`	Notify	Acknowledgment flag
`Sensor Data`	`555a0002-003...`	Notify	Sends sampled sensor data
`Timestamp`	`555a0002-0034...`	Write	Receives the UNIX timestamp
`Campaign Config`	`555a0002-0035...`	Write	Receives sampling rate and counter
`Activity Cmd`	`555a0002-0040...`	Write	Start / Stop / Sleep / End
`Battery Service`	`180F`	Notify/Read	Standard battery level service

Table 1: BLE Characteristics and their Roles

Each characteristic is associated with a human-readable descriptor defined using the BLE standard descriptor UUID '0x2901'. This allows BLE central devices to identify the role of each characteristic:

Listing 2: BLE Descriptors

```
BLEDescriptor ackDescriptor("2901", "ACK");
BLEDescriptor sensorDescriptor("2901", "SENSOR_DATA");
BLEDescriptor timestampDescriptor("2901", "TIMESTAMP");
BLEDescriptor campaignDescriptor("2901", "CAMPAIGN");
BLEDescriptor activityDescriptor("2901", "ACTIVITY");
```

This structure ensures a clear separation of concerns between commands, data, and status updates, while maintaining BLE GATT compliance and allowing easy pairing with BLE central applications.

12.2 Write / Notify / Read Communication Pattern

The communication between the BLE central and the peripheral board follows the **Write / Notify / Read** pattern, a standard approach in Bluetooth Low Energy (BLE) applications that allows clear separation of roles and efficient data exchange.

Write Operations

Write operations are used by the central to send information or commands to the peripheral. These are handled via characteristics with the `BLEWrite` property.
In this project, the following writeable characteristics are defined:

- **Timestamp** – used to set the reference time on the board.
- **Campaign Configuration** – used to configure sampling parameters such as initial counter and frequency.
- **Activity Command** – used to control the flow of the acquisition (e.g., `START`, `STOP`, `SLEEP`, `END`).

Notify Operations

Notify operations are initiated by the peripheral to send data to the central without requiring explicit polling. The central must subscribe to these notifications to receive them.
The following characteristics use the `BLENotify` property:

- **ACK** – communicates acknowledgment codes (e.g., `OK`, `KO`) in response to control messages.
- **Sensor Data** – transmits packets of sampled accelerometer data at a configured frequency.
- **Battery Level** – notifies the central when a new battery reading is available.

Read Operations

Read operations allow the central to explicitly request the current value of a characteristic. While less efficient for streaming data, this is useful for on-demand values such as status or configuration.
The only characteristic using the `BLERead` property in this project is:

- **Battery Level** – allows reading the current charge percentage of the Li-Po battery.

Summary Table

Characteristic	**Direction**	**Purpose**
`Timestamp`	Write	Set initial time from central
`Campaign Configuration`	Write	Configure sampling parameters
`Activity Command`	Write	Control acquisition flow
`ACK`	Notify	Send response codes to central
`Sensor Data`	Notify	Send accelerometer samples
`Battery Level`	Notify / Read	Periodic or requested battery info

Table 2: Summary of BLE communication patterns per characteristic

13 Data Acquisition & Processing

This section describes how data acquisition is handled on the peripheral device, focusing on the configuration of the Inertial Measurement Unit (IMU), the sampling strategy, and the chosen data format for efficient transmission and storage.

13.1 IMU Configuration and Usage

The board is equipped with an LSM9DS1 9-axis IMU, integrated on the Arduino Nano 33 BLE Sense board. This sensor includes a 3-axis accelerometer, gyroscope, and magnetometer. However, in this project, only the accelerometer is used to collect motion data.
The sensor is initialized at startup using the `Arduino_LSM9DS1_AAAC` library, and its availability is checked to ensure proper operation. The accelerometer readings are acquired via the `IMU.readAcceleration()` function inside the acquisition loop.

13.2 Sampling Rate and Precision

The sampling rate is dynamically configured via the BLE central by writing to the `Campaign Configuration` characteristic. The accepted frequency is included in the payload and translated into a sampling interval expressed in milliseconds.
The relation is computed as:

$$\texttt{sampling\_interval (ms)} = \frac{\texttt{SAMPL\_FREQ\_ADJ} \cdot 1000}{\texttt{sampling\_frequency (Hz)}}$$

where `SAMPL_FREQ_ADJ` compensates for internal timing discrepancies and is defined as a constant.
Each sample is timestamped indirectly using an internal counter and includes three floating point values corresponding to the X, Y, and Z axes of the accelerometer.

13.3 RAW Format

Sensor data are structured into a packed format before being written to the SD card and sent via BLE notification. Each data packet (`Datapacket_t`) consists of:
This raw format ensures that all acquired data are preserved without filtering or compression, enabling post-processing and analysis on the receiving device. Before being transmitted, data are optionally printed in hexadecimal format for debugging or logged as CSV-like plain text to the SD card.

14 Data Logging

The device supports persistent data logging by saving all sampled values to a microSD card during acquisition. This feature allows post-experiment analysis even when the BLE connection is unstable or temporarily lost.

14.1 SD Card Integration

The SD card module is connected to the microcontroller via the SPI interface. Initialization occurs at startup, and the module is configured using the standard `SD` and `SPI` libraries provided by Arduino. If the SD card fails to initialize, the device enters a blocking state to avoid data loss or undefined behavior.

14.2 File Format and Naming

Each session log is stored in a file named `Data_N.txt`, where `N` is an incremented integer. The contents of the file follow a specific structure:

- The **first line** contains the external timestamp received from the BLE central, in ISO 8601 format, e.g.:

  ```
  2025-07-17T15:43:12.000
  ```

- Each **subsequent line** represents a data sample and is prefixed by the internal timestamp (milliseconds since boot), followed by a hexadecimal string encoding the measurement data:

  ```
  2025-07-17T15:43:12.250 0x000503F1FFC3001A
  2025-07-17T15:43:12.500 0x000603E9FFE2000C
  ```

Each hex value is serialized in little-endian order, and each byte is printed as a 2-digit hex number, including leading zeros.

14.3 Hex Logging Format

Bytes	Field	Description
2	Counter	16-bit unsigned integer (default_counter)
2	X	16-bit signed integer (X-axis acceleration)
2	Y	16-bit signed integer (Y-axis acceleration)
2	Z	16-bit signed integer (Z-axis acceleration)

Table 3: Structure of each logged sample payload (hex-encoded)

The addition of the timestamp provides a clear temporal reference for each sample, which enables reconstruction of the signal timeline even when samples are not collected at a fixed interval due to system delays or BLE congestion.

15 Connection Management

Reliable connection management is essential to ensure data consistency and user control during the acquisition process. The system uses a combination of BLE polling, timeout detection, and finite state transitions to manage connectivity and ensure graceful behavior in the case of disconnections or user-initiated stops.

15.1 Timeout Handling

A disconnection timeout mechanism is used to detect when the central device (e.g., a smartphone or PC) is no longer reachable. After the connection is lost, a countdown begins. If the board does not re-establish the BLE connection within a predefined period (defined in the configuration file), it assumes the session has failed irreversibly.

Once the timeout is exceeded:

- The board enters the **Final** state.
- The SD file is closed safely to prevent data loss.
- All acquisition flags and internal variables are reset.
- The system halts to avoid undefined behavior.

15.2 Reconnection Strategy

During a disconnection, the system temporarily enters the **Disconnection** state. While in this state:

- The device keeps advertising itself to allow a BLE central to reconnect.
- If reconnection occurs within the timeout period, and the campaign was previously ongoing, the system returns to the **Sleep** state and waits for further instructions.
- This allows the user to either restart the campaign or terminate it manually without data loss.

This strategy minimizes campaign interruptions and allows recovery from transient connection issues without requiring a device reset.

15.3 Connection Management Overview

The overall connection management flow is handled within the main `loop()` function using:

- Periodic polling with `BLE.poll()` to detect connection status changes.
- A state machine that governs transitions between `Power_On`, `Wait_TimeSt.`, `Acquisition`, `Disconnection`, and `Final`.
- A configurable timeout mechanism that ensures reliable fault detection.

Connection transitions and related behaviors are clearly illustrated in the FSM diagram (Figure 1), where reconnection paths and failure paths are highlighted in blue and red, respectively.

16 Battery Monitoring

The device integrates a basic battery monitoring system to track the current voltage level of the LiPo battery and make this information accessible to the user via BLE. This helps in preventing unexpected shutdowns and in ensuring safe operation.

16.1 ADC Setup

Battery voltage is monitored using the internal Analog-to-Digital Converter (ADC) of the board. A dedicated analog pin is connected to a voltage divider circuit to bring the battery voltage within the ADC input range (typically 0–3.3V). The ADC periodically reads the battery voltage, which is then converted to a percentage representation using a calibration formula based on the LiPo voltage range (typically 3.0V to 4.2V).

The measurement is performed as part of the main acquisition loop or at regular intervals during idle states, depending on power management considerations.

16.2 BLE Battery Characteristic

To expose the battery level to the BLE central device, the board implements the standard BLE Battery Service using the UUID `180F` and the Battery Level Characteristic with UUID `2A19`. This characteristic supports both `BLERead` and `BLENotify` operations.
The battery level is stored as a single byte (0–100), representing the percentage of remaining capacity.
Battery voltage is read from the analog pin A0 and mapped to a 0–100% value using the standard map() function. The function responsible for reading the analog input and updating the BLE characteristic is:
This function is typically called at regular intervals during device operation, or at important lifecycle stages (e.g., startup, campaign start/end), in order to keep the central updated on the battery status.

17 Configuration Parameters

This system supports two primary methods of configuration: an initial static configuration via a `JSON` file stored on the SD card, and a runtime configuration via BLE communication. These two approaches provide flexibility during development and deployment.

17.1 JSON file (`config.json`) structure

Upon startup, the device reads a `config.json` file from the SD card to load initial parameters such as sampling frequency, campaign duration, and disconnection timeout. This static configuration allows the system to operate even in the absence of immediate BLE communication. An example of the `config.json` structure is shown below:

```
{
  "default_counter": 0,
  "default_freq": 50,
  "default_timer": 60,
  "default_lost_connection_timeout": 10
}
```

- **default_counter**: Initial value for the campaign's sample counter.
- **default_freq**: Default IMU sampling frequency in Hz.
- **default_timer**: Default campaign duration in seconds.
- **default_lost_connection_timeout**: Number of seconds before triggering a Final state if connection is lost.

If the file is missing or corrupted, the device will remain in a wait state to avoid unsafe behavior.

17.2 Runtime BLE configuration

After establishing a BLE connection, the central can dynamically configure the device by writing specific values to dedicated BLE characteristics. This enables interactive configuration without requiring an SD card modification.

The following runtime parameters can be set:

- **Timestamp** (`UUID_CHR_TIMEST`): The payload must be 4 bytes representing the epoch timestamp.
- **Campaign configuration** (`UUID_CHR_CAMPCONF`): Contains initial counter and sampling frequency (Hz). This characteristic accepts either a 3-byte payload (counter LSB, MSB + frequency) or a confirmation byte (1-byte).
- **Activity control** (`UUID_CHR_ACTIVITY`): Used to start or stop the data acquisition. Accepts commands like `START_CAMPAIGN`, `STOP_CAMPAIGN`, `SLEEP_CAMPAIGN`, and `END_PROCESS`.

This dynamic configuration flow is designed to be safe and state-aware: for instance, the configuration payload is accepted only after receiving a valid timestamp.

18 How to use

18.1 Steps to start the device

To properly initialize and operate the device, follow the steps below:

1. Power on the device. The system will enter the `Power_On` state and wait for a BLE central connection.
2. Connect to the device using a compatible BLE central (e.g., smartphone, PC).
3. Subscribe to every notification characteristic to receive updates:
   - `ACK` for acknowledgment messages.
   - `Sensor Data` for accelerometer samples.
   - `Battery Level` for periodic battery updates.
4. Write the timestamp to the `Timestamp` characteristic.
5. Configure the campaign by writing the campaign parameters to the `Campaign` characteristic.
6. Wait for the device to acknowledge the configuration by notifying an `ACK`.
7. Start the acquisition process by writing the `START` command to the `Activity` characteristic.
8. Optionally, adjust the sampling frequency at runtime by sending a `FREQ` message.
9. To end the acquisition, write a `STOP` or `END` command to the `Activity` characteristic.

18.2 BLE client interaction

The BLE client is expected to interact with the device by writing and subscribing to specific characteristics:

- **Write** operations:
  - `Timestamp`: send a 4-byte Unix timestamp (hex).
  - `Campaign`: send campaign configuration (init counter and sampling frequency).
  - `Activity`: control the campaign lifecycle (START, STOP, SLEEP, END).
- **Subscribe to notifications** on:
  - `Sensor data`: receives raw accelerometer samples.
  - `ACK`: receives acknowledgment of correct configuration.
  - `Battery level`: receives periodic battery level updates.
- **Optional read**:
  - `Battery level`: the client can read this value at any time.

All communications must respect the expected payload size and order of operations to ensure correct system behavior.

19 Possible Improvements

Several enhancements could be considered for future versions of the project:

- Introduce encryption and authentication mechanisms for BLE communication.
- Add support for dynamic reconfiguration of parameters during acquisition.
- Improve power-saving strategies by managing sensor activation and BLE advertising more efficiently.

20 Appendices

20.1 Example Data

An example of the data logged in the SD file is as follows:

```
2025-07-01T12:42:33
0001DE10FF1A
0001DE25FF19
...
```

Each line represents a dataPacket_t encoded in hexadecimal, including: default_counter, x, y, z.

20.2 BLE Commands Overview

Command	**Characteristic**	**Description**	**Payload**
`Timestamp`	`TIMESTAMP`	Sends a UNIX timestamp to synchronize the board	0x12345678
`Campaign Config`	`CAMPAIGN`	Sets counter and frequency (3-byte payload)	0x000001
`FREQ Update`	`CAMPAIGN`	Updates sampling frequency dynamically (2-byte)	0xAA
`START_CAMPAIGN`	`ACTIVITY`	Begins the data acquisition process	0x00
`STOP_CAMPAIGN`	`ACTIVITY`	Terminates acquisition and stores data to SD	0x0F
`SLEEP_CAMPAIGN`	`ACTIVITY`	Puts the device into sleep mode	0xFF
`END_PROCESS`	`ACTIVITY`	Finalizes campaign and transitions to Final state	0xBB

Table 4: BLE command structure and meaning

21 Central Part

22 Software Overview

The software is a cross-platform Python application designed to manage Bluetooth Low Energy (BLE) communication with multiple Inertial Measurement Units (IMUs). It allows users to configure, control, and collect real-time data from IMU devices, supporting both CSV-based local storage and MQTT-based cloud transmission.

22.1 External Libraries Used

The software relies on several external Python libraries to provide BLE communication and optional data transmission. Below is a summary of the main libraries used:

- **Bleak** (`bleak`): A cross-platform BLE (Bluetooth Low Energy) library for Python. It provides asynchronous APIs for scanning, connecting, reading, writing, and subscribing to BLE characteristics. Bleak supports Windows (via WinRT), macOS (via CoreBluetooth), and Linux (via BlueZ).
- **Paho MQTT** (`paho.mqtt.publish`): A client library for the MQTT protocol developed by the Eclipse Foundation. It is used to publish sensor data to a remote MQTT broker for cloud-based storage or real-time processing.

22.2 Main Functions Overview

This section provides a detailed overview of the primary functions implemented in the code, describing their purpose, behavior, return values, and typical invocation context.

- `sensor_collect_callback_wrapper(name: str)`
  This function returns a callback function designed to handle incoming sensor data notifications from a specific IMU identified by `name`. The inner callback unpacks the raw byte data into meaningful sensor values (counter and accelerations), formats them as a CSV string, and passes the result to the configured storage function (e.g., MQTT publishing or CSV buffering). It also sets the IMU as ready if it was previously marked as not ready. This callback is invoked asynchronously upon receiving sensor data notifications from the BLE characteristic `UUID_CHR_SENSOR`.
- `ack_callback_wrapper(name: str)`
  Returns a callback function that handles acknowledgment (ACK) messages from the IMU. When an ACK with value `ACK_OK` is received, the function sets the corresponding IMU as ready by setting an asynchronous event. This callback is triggered upon notifications from the characteristic `UUID_CHR_ACK`.
- `battery_level_callback_wrapper(name: str)`
  Provides a callback function to handle battery level notifications from an IMU. Currently, it simply prints the raw data received. It is called upon notifications on the `UUID_NOTIFY_BATTERY` characteristic.
- `disconnection_callback_wrapper(name: str)`
  Returns a callback function that is invoked when the BLE client disconnects unexpectedly from an IMU device. It logs a warning, removes the device from the available IMU list, and, if the system is not shutting down, triggers a reconnection attempt asynchronously.

- `async def reconnect(name: str) -> bool`
  Handles the process of rediscovering and reconnecting to an IMU device identified by `name`. It scans for the device, attempts connection, configures the campaign if necessary, and starts data acquisition. Returns `True` if reconnection and setup succeed, otherwise `False`. Invoked when a disconnection occurs or during initial setup retries.

- `async def discover(name: str) -> bool`
  Scans for a BLE device with the given `name` within a timeout window. Upon successful discovery, creates a BLE client and attempts connection by calling `connect`. Returns `True` if the device is found and connected, `False` otherwise. Called during startup or reconnection.

- `async def connect(name: str, client: BleakClient) -> bool`
  Establishes a BLE connection to the given client, subscribes to relevant BLE characteristics for sensor data, acknowledgments, and battery notifications. Sets the IMU ready event to false initially. Returns `True` if the connection and subscriptions are successful. Called by `discover`.

- `async def setup_campaign(name: str, reconn=False) -> bool`
  Ensures that the IMU is configured with the correct campaign parameters and is marked as ready before data acquisition. If the IMU is not ready, sends timestamp and campaign configuration packets and waits for an acknowledgment. Allows up to 2 attempts before failing. Returns `True` on success, `False` on failure. Invoked after connection and during reconnection attempts.

- `async def send_timestamp(name: str) -> None`
  Sends the current timestamp to the IMU as a configuration parameter. Called during campaign setup.

- `async def send_campaign_parameters(name: str) -> None`
  Sends the campaign initialization counter and sampling frequency parameters to the IMU. Called during campaign setup.

- `async def send_sampling_frequency(name: str, freq: int = 1) -> None`
  Updates the sampling frequency of the specified IMU dynamically during runtime. Called in response to runtime commands.

- `async def start(name: str) -> None`
  Sends the `START_CAMPAIGN` command to the IMU, putting it into **Acquisition** state. Called upon the "start" runtime command.

- `async def stop(name: str) -> None`
  Sends the `STOP_CAMPAIGN` command, setting the IMU into **Wait_Start** state. Called upon the "stop" runtime command.

- `async def sleep(name: str) -> None`
  Sends the `SLEEP_CAMPAIGN` command, putting the IMU into low-power **Sleep** state. Called upon the "quit" runtime command.

- `async def end(name: str) -> None`
  Sends the `END_CAMPAIGN` command, signaling the **Final** state of the IMU. Called upon the "shutdown" runtime command.

- `async def disconnect(name: str) -> None`
  Gracefully disconnects the BLE client from the IMU device. Called during safe shutdown procedures.

- `async def safe_close() -> None`
  Initiates a safe shutdown procedure by signaling termination, disconnecting all devices, and flushing any buffered data (e.g., CSV buffers). Called when the program exits or receives shutdown commands.

- `def store_mqtt(data: str) -> None`
  Publishes data to the configured MQTT broker and topic. Used as a storage callback if MQTT is selected as the storage method.

- `def store_csv(data: str) -> None`
  Buffers incoming data into an in-memory list and periodically flushes it to a CSV file to minimize memory usage. Used as a storage callback if CSV is selected as the storage method.

23 Connection/Reconnection Logic

When an IMU is detected by the connection mechanism, it can either be connecting for the first time or have been previously connected. In the latter case, it is necessary to distinguish between two situations:

1. **IMU requires configuration**
2. **IMU already configured**

To determine which condition applies, as soon as the IMU connects, the system waits for the IMU to be set as "ready" through the `sensor_collect_callback`, which is triggered upon receiving data from the sensor.

If no data is received within a timeout equal to:

$$T_{\text{timeout}} = \frac{1}{f_s} + 1 \text{ second}$$

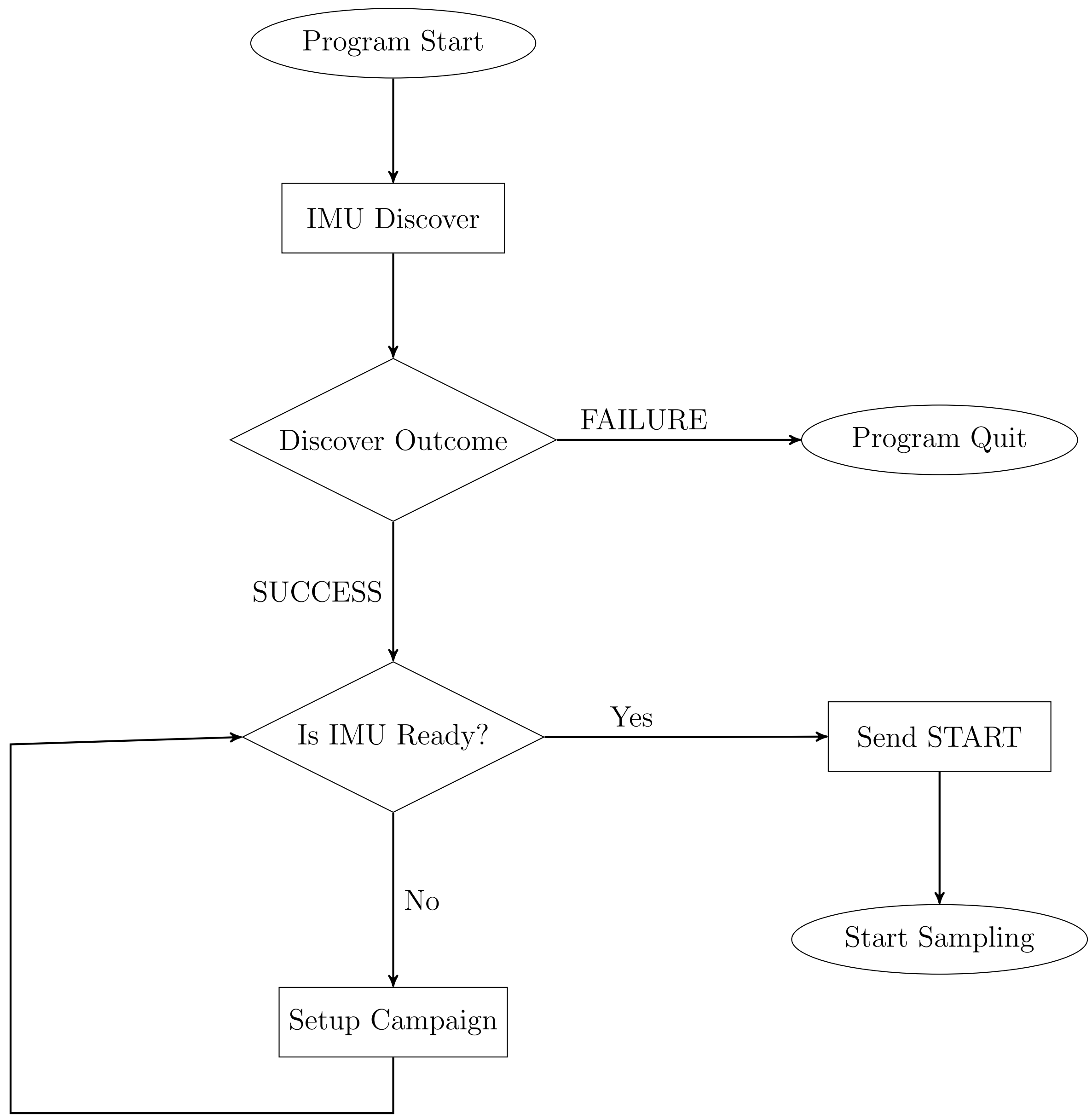
where f_s is the sampling frequency, the system assumes that the IMU is currently **not configured**.
At this point:

- If it is the first time that the specific IMU connects, the system proceeds with its configuration by sending the necessary parameters to start the acquisition campaign.
  After sending the configuration, the system waits for the IMU to signal readiness via the `ack_callback`, triggered only if the configuration is valid.

- If the IMU had already been connected before and was expected to be transmitting data, the lack of received data indicates that the IMU was simply reset during transmission. In this case, to avoid counter misalignment among different IMUs (which could compromise the integrity of the collected data), the system closes the connection and terminates the process.

This logic ensures that the acquired data is synchronized across all IMUs and prevents inconsistencies caused by uncoordinated resets or restarts of the devices.

In the following, a flow diagram is provided that illustrates this logic to clarify the decision-making process during the IMU connection.



For the reconnection mechanism, the procedure is the same; however, if the IMU is judged to be not ready (i.e., not configured), the program execution is terminated.

24 Data Storage Methods

The system provides flexible mechanisms for storing the sensor data collected from the IMUs, supporting two primary modes: **CSV-based local file storage** and **MQTT-based remote publishing**. The storage method is defined in the configuration file via the `store_method` field and dynamically determines how incoming sensor data is handled during runtime.

CSV Storage. When `store_method` is set to `csv`, the system writes data to a local file on disk. To reduce disk I/O and avoid frequent write operations, an in-memory buffer is used. This buffer collects sensor records up to a configurable limit, defined by the `CSV_BUFFER_SIZE` constant. Once the buffer reaches this limit, the contents are flushed to disk.

Each CSV file is automatically named based on the current date and a progressive index to prevent overwriting previous logs. For example, if the acquisition takes place on July 24, 2025, the generated filenames will follow the pattern `20250724_01.csv`, `20250724_02.csv`, and so on. This systematic naming ensures traceability and chronological organization of multiple recording sessions performed on the same day.

This method is particularly useful for offline data acquisition or when post-processing is required.

Data is stored in CSV format with the following column structure:

```
Timestamp,IMU,Counter,Acceleration X,Acceleration Y,Acceleration Z
1753268493,1-IMU,0,1002,-2896,7788
1753268493,2-IMU,0,-288,403,7867
1753268495,1-IMU,1,1325,-3190,7263
1753268495,2-IMU,1,-292,402,7869
1753268497,1-IMU,2,1408,-3151,7270
1753268497,2-IMU,2,-292,410,7860
1753268499,2-IMU,3,-294,403,7859
1753268499,1-IMU,3,1477,-3111,7274
1753268501,1-IMU,4,1552,-3074,7298
1753268501,2-IMU,4,-297,407,7881
                ...
```

MQTT Publishing. If the storage mode is set to `mqtt`, the system uses the MQTT protocol to publish each sensor reading to a configured broker. The parameters `mqtt_broker`, `mqtt_port`, and `mqtt_topic` are specified in the configuration file. Each IMU reading is sent as a plain string message. This mode is designed for real-time data transmission in distributed or cloud-connected systems, where immediate processing or visualization is required.

Standard Output Fallback. If an unsupported or missing storage mode is specified, the system defaults to printing sensor data to the standard output. This fallback mode is useful for debugging or when minimal configuration is preferred.

Design Considerations. Both storage mechanisms are abstracted through a common function pointer (`STORE_FUNC`), allowing the acquisition logic to remain decoupled from the storage backend. Additionally, when stopping data acquisition, the system ensures that buffered data is safely flushed to disk (in CSV mode) before shutting down to prevent data loss.

25 Runtime Commands from Terminal

During execution, the program enters an interactive loop waiting for user input via standard input. The following runtime commands are supported:

`help` Displays a list of available commands with brief descriptions.

`start` Initiates data acquisition on all connected IMUs by sending the `START_CAMPAIGN` command. In CSV mode, it opens (or reopens) the current CSV file for appending data. All IMUs transition to the **Acquisition** state.

`stop` Stops data acquisition by sending the `STOP_CAMPAIGN` command to each IMU. In CSV mode, this also triggers flushing of the buffer to disk. All IMUs transition back to the **Wait_Start** state.

`freq <n>` Updates the sampling frequency of all IMUs. The argument n must be an integer in the range 0-255. The system applies the new frequency immediately by sending the `FREQ` command via BLE.

`shutdown` Sends the `END_CAMPAIGN` command to all IMUs, requesting them to gracefully terminate the current acquisition campaign. All IMUs transition to the **Final** state. The program then exits the interactive loop and proceeds to a clean shutdown.

`quit` Sends the `SLEEP_CAMPAIGN` command to all IMUs, placing them into low-power sleep mode. All IMUs transition to the **Sleep** state. The program then exits the interactive loop and performs a clean shutdown.

Usage Example. Upon running the script, the user is presented with a prompt like "`>> `". Typing one of the above commands executes the corresponding action immediately, without restarting the program. For example:

```
>> help
Available commands:
  help      Show this help message
  start     Start data acquisition (Acquisition state)
  stop      Stop data acquisition (Wait_Start state)
  freq n    Set sampling frequency to n (0-255)
  shutdown  End campaign and exit (Final state)
  quit      Sleep IMUs and exit (Sleep state)
```

Note

All commands are broadcasted to the IMUs in parallel to ensure synchronized execution across devices. This parallelism helps to maintain temporal alignment in data acquisition.

26 External Configuration

The system supports an external configuration mechanism that allows users to specify operational parameters prior to runtime. This configuration is provided through a JSON file, which defines essential settings such as the list of IMUs to be managed, connection timeouts, initial counter values, sampling frequencies, and data storage preferences.

Upon startup, the software reads the configuration file and initializes its internal parameters accordingly.

Key configurable parameters include:

- **IMU Identifiers:** A list of IMU device names or addresses that the system will attempt to discover and connect with.
- **Timeout:** Defines the maximum time allowed for device discovery and connection attempts, ensuring the system remains responsive and does not hang indefinitely.
- **Initial Counter:** Sets the starting value for the internal data counters used in campaign synchronization, crucial for data alignment across multiple IMUs.
- **Sampling Frequency:** Specifies the rate at which each IMU collects and transmits sensor data, influencing both data granularity and communication load.
- **Data Storage Method:** Determines whether data are stored locally in CSV files or published remotely via MQTT. The supported values are `"csv"` and `"mqtt"`. Any other value will be ignored, and the system will default to printing data to the standard output.
- **CSV Storage Directory:** Specifies the folder path where CSV files will be saved.

The configuration file structure is validated at startup to detect missing or malformed parameters, with appropriate error messages guiding users to correct issues before the system proceeds.

An example of a correct configuration file is as follows:

```
{
  "imus": ["1-IMU", "2-IMU"],
  "timeout": 300,
  "init_counter": 0,
  "sampling_frequency": 1,
  "store_method": "csv",
  "mqtt_broker": "131.175.*.*",
  "mqtt_port": 1883,
  "mqtt_topic": "aaac/campaign/imu",
  "csv_store_dir": "csv"
}
```

27 How to Use

27.1 Installation

Before running the system, it is necessary to have Python version 3.8 or higher installed on your machine.

In the root directory of the project, you will find a file named `requirements.txt`, which lists all the required Python libraries.

To install the dependencies, run the following command:

```
pip install -r requirements.txt
```

Before launching the application, make sure that the configuration file is correctly defined. For a detailed explanation of the configuration structure and an example, refer to Section 26.

A sample configuration file named `collect_hub_config.json` is already provided in the root directory of the project and can be used as a base or directly adapted to your needs.

27.2 Startup

Once the dependencies are installed and the configuration file is properly set, you can start the application from the command line.

The program requires the path to the configuration JSON file as a command-line argument. For example, to run the application using the provided sample configuration file, execute:

```
python central.py collect_hub_config.json
```

During startup, the program immediately begins scanning for the IMUs listed in the configuration file. As soon as a device is found, the system connects to it and proceeds to configure it according to the specified configuration parameters.

This is an example of the output displayed once the program is started:

```
[info] found device: 2-IMU
[info] found device: 1-IMU
[info] connected to 2-IMU
[info] connected to 1-IMU
[info] 2-IMU is not configured
[info] 1-IMU is not configured
[info] sent timestamp to 1-IMU: 1753368592
[info] sent timestamp to 2-IMU: 1753368592
[info] sent campaign parameters to 1-IMU: {'init_counter': 0, 'sampling_frequency': 1}
[info] sent campaign parameters to 2-IMU: {'init_counter': 0, 'sampling_frequency': 1}
>>
```

At this point, all IMUs are correctly configured, and it is possible to proceed with starting the sampling and receiving data.

28 Example Scenarios and Program Output

28.1 Starting Data Acquisition

Using the `start` command, the program initiates the sampling on all configured IMUs. The system sends the `START` command to each device, indicated by the output below:

```
>> start
[info] sent START to 1-IMU (0x00)
[info] sent START to 2-IMU (0x00)
```

28.2 Stopping Data Acquisition

To stop the ongoing data acquisition, the `stop` command is used. The program sends the `STOP` command to all IMUs, as shown in the output:

```
>> stop
[info] sent STOP to 1-IMU (0x0F)
[info] sent STOP to 2-IMU (0x0F)
```

28.3 Closing the Program with `quit`

The `quit` command puts all IMUs into `SLEEP` state and disconnects the devices safely. The output reflects the commands sent and the disconnections detected:

```
>> quit
[info] sent SLEEP to 1-IMU (0xFF)
[info] sent SLEEP to 2-IMU (0xFF)
[warning] device 1-IMU disconnected
[warning] device 2-IMU disconnected
```

28.4 Closing the Program with `shutdown`

Alternatively, the `shutdown` command is used to terminate the program by sending the `END` command to all IMUs before disconnecting:

```
>> shutdown
[info] sent END to 2-IMU (0xBB)
[info] sent END to 1-IMU (0xBB)
[warning] device 2-IMU disconnected
[warning] device 1-IMU disconnected
```

28.5 Disconnection Due to Device Out of Range

In case an IMU disconnects because it moves out of range, the program logs a warning and continues scanning to reconnect the device automatically. Once the device is found again, the system reconnects and can resume the acquisition, as shown below:

```
>>
[warning] device 2-IMU disconnected
[info] found device: 2-IMU
[info] connected to 2-IMU
>>
```

28.6 Disconnection Due to IMU Reset

When an IMU reconnects, the system is able to determine whether the device has been reset or not. If a reset is detected, the program logs a warning and initiates a safe shutdown procedure.

```
>>
[warning] device 1-IMU disconnected
[info] found device: 1-IMU
[info] connected to 1-IMU
[warning] 1-IMU has been reset. Quitting...
[info] press ENTER to exit
[info] sent SLEEP to 2-IMU (0xFF)
[info] sent SLEEP to 1-IMU (0xFF)
[warning] device 2-IMU disconnected
[warning] device 1-IMU disconnected
```

28.7 Connection Hang After IMU Reset

In some cases, immediately after an IMU has been reset, the BLE library `bleak` is able to detect the device's advertising packet but then gets stuck during the connection attempt. This situation causes the connection process to hang indefinitely.

To mitigate this issue, a timeout mechanism of 5 seconds has been implemented. If the connection is not established within this timeout, the attempt is aborted and the following error message is displayed:

```
[info] found device: 1-IMU
[error] 1-IMU unavailable. Please try again later.
```

To resolve this issue, it is usually sufficient to reset the affected IMU and restart the program.