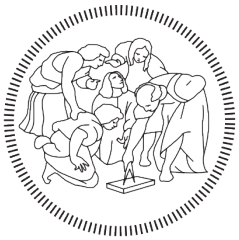
BLE-BASED MOTION DATA LOGGER

Report

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 Introduction

1.1 Project Title

BLE-based Motion Data Logger for IoT Applications

1.2 Team Members and Roles

This project was carried out by a team of two students:

- **Alessandro Albrigo**: In charge of the embedded software development, BLE protocol design and firmware integration on the Arduino Nano 33 BLE platform.
- **Giuseppe Boniver Conte**: Responsible for the development of the BLE central client in Python, the design of the configuration workflow, and the data logging infrastructure.

1.3 Duration

The development of this project spanned approximately six months, from March to August 2025, including design, implementation, testing, and documentation phases.

1.4 Overview and Goals

The project stems from the need to modernize and extend the capabilities of an existing IoT device that collects motion data using an Inertial Measurement Unit (IMU). The legacy system was based on a microcontroller communicating via the ANT protocol with a base station. While functional, the original solution suffered from three major limitations:

1. **Limited compatibility**: ANT is not natively supported on most consumer devices like laptops or smartphones.
2. **Low energy efficiency**: Due to the lack of sleep modes and persistent polling.
3. **Poor scalability and maintainability**: Caused by monolithic firmware and tight coupling between components.

To address these issues, the project introduced a reengineering of both the firmware and communication architecture, based on Bluetooth Low Energy (BLE). The new implementation allows the device to function as a configurable, low-power data acquisition platform, capable of transmitting data in real time and storing it locally on a microSD card.

Main objectives:

- Replace the ANT communication with BLE 4.0 to increase device interoperability.
- Design a modular firmware based on a Finite State Machine (FSM) for robustness and clarity.
- Allow runtime configuration and monitoring via a Python-based client.
- Support offline logging for reliable data acquisition in scenarios with intermittent connectivity.
- Enable battery status monitoring and graceful disconnection recovery.

Target applications of the device include:

- Wearable sensors for sports and rehabilitation.
- Motion analysis platforms for robotics and biomechanics.
- Multi-sensor synchronized data collection for research purposes.

This document outlines the design choices, challenges, solutions and outcomes of the project, with detailed discussions of both hardware and software components, communication protocols, user workflows, and final results.

2 Related Work

2.1 Existing Solutions

Many modern fitness trackers and motion analysis systems rely on BLE to communicate IMU data to smartphones and PCs. Examples include commercial devices such as Fitbit, Apple Watch, and open-source platforms like MetaMotion or Bluefruit Feather from Adafruit.

2.2 Comparative Analysis

Compared to these, our system:

- Uses a flexible and reconfigurable BLE protocol tailored to the application.
- Supports raw data acquisition for offline processing.
- Allows for fully autonomous use thanks to onboard SD logging.

3 Innovation of the Project

3.1 Novelty and Improvements

This project introduces several innovations in both the design and implementation of the IoT system compared to existing motion tracking solutions.

1. Custom BLE GATT Protocol Unlike generic BLE applications, this project defines a tailored set of GATT services and characteristics, enabling fine-grained control of the device via commands such as `START`, `STOP`, `SLEEP`, and `END`. Each command corresponds to a specific state in the internal FSM and allows the user to interact with the device without requiring reprogramming or restarting.

2. Modular FSM-Based Firmware The firmware is fully driven by a finite state machine (FSM), where each state represents a precise stage in the acquisition campaign. Transitions are triggered by BLE events or internal timeouts. This provides:

- Robustness against communication failures.
- Clear separation between operational phases.
- Maintainable and testable code.

3. Dual-mode Data Logging Motion data is transmitted via BLE in real time and simultaneously saved to a microSD card in raw hexadecimal format. This hybrid design ensures that no data is lost even when BLE connectivity is interrupted and allows for offline analysis.

4. Runtime Configuration via BLE The device accepts BLE messages to set acquisition parameters such as sampling frequency, initial counter, and runtime timestamping. This avoids the need to modify source code or recompile firmware for different experiments.

5. BLE Battery Monitoring A voltage divider is used to measure battery voltage through an analog pin. The value is converted to a percentage and exposed via a BLE Battery Service, offering real-time monitoring and safe shutdown in case of low power.

3.2 Benefits and Broader Impact

- **Portability**: The system can be used on-the-go, without requiring constant connection to a PC or cloud server.
- **Scalability**: Multiple devices can be managed from a single central application.
- **Extensibility**: The firmware structure supports adding new services (e.g., gyroscope, magnetometer, GPS).
- **Cross-platform support**: The Python client leverages `bleak`, a cross-platform BLE library that supports Linux, macOS, and Windows.
- **Open-source potential**: The project can be easily adapted by students, researchers or hobbyists who need a customizable BLE-based motion tracker.

By integrating modern communication standards, modular design patterns, and robust data handling mechanisms, the system provides a solid foundation for future developments in the domain of wearable sensing and low-power IoT solutions.

4 Project Development

4.1 Software and Tools Used

The development of the BLE-based IoT data logger system involved a range of hardware and software tools across multiple platforms. On the embedded side, the Arduino Nano 33 BLE board with the Nordic nRF52840 microcontroller was used as the primary platform. Firmware development was carried out in C/C++ using the Arduino IDE and PlatformIO. BLE client development was done in Python using Visual Studio Code.

Embedded Environment

- **Arduino IDE / PlatformIO**: Used for writing, compiling, and flashing firmware to the Arduino Nano 33 BLE.
- **Arduino Libraries**:
  - `ArduinoBLE.h`: BLE communication support.
  - `SPI.h`, `SD.h`: Communication with the microSD card.
  - `Arduino LSM9DS1 AAAC.h`: IMU sensor interface.
  - `ArduinoJson.h`: JSON parsing for config files.
  - `TimeLib.h`: Handling timestamps.

Python BLE Client

- **Python 3.10+**: Programming language for the central BLE application.
- **bleak**: Cross-platform BLE library for scanning, connecting, and communication.
- **paho.mqtt.publish**: Used optionally to send data to MQTT brokers.
- **asyncio**: For asynchronous BLE communication and command processing.

Version Control and Documentation

- **Git / GitHub**: Source code versioning, collaboration, and issue tracking.
- **LaTeX**: Used for producing this documentation.

4.2 Development Phases

The project followed an iterative and modular development process. The following phases were identified:

1. **Requirement Analysis**: This phase focused on reverse engineering the legacy system, originally based on the ANT protocol. We analyzed the structure and limitations of the firmware and communication model, identifying pain points such as the lack of extensibility, the complexity of pairing, and poor cross-platform support.

2. **Firmware Architecture Redesign**: A new architecture was proposed based on a Finite State Machine (FSM). Each operational step of the firmware—booting, connecting, configuring, sampling, logging, and sleeping—was mapped to a state, with clearly defined transitions. The redesign aimed to improve modularity, fault tolerance, and power efficiency.

3. **BLE Protocol Implementation**: We replaced the ANT protocol with BLE by defining a custom GATT service that included characteristics for configuration, timestamping, and data transmission. BLE UUIDs were chosen using a consistent naming pattern, and BLE descriptors were added to make the services human-readable when using BLE exploration tools.

4. **Data Logging Infrastructure**: A logging module was added to the firmware to write raw accelerometer data to an SD card in hexadecimal format. The file format was designed to include both ISO 8601 timestamps and raw binary-encoded samples, allowing precise reconstruction of data acquisition sessions.

5. **Client-side Application**: A Python application was developed using the `bleak` library to serve as the BLE central. This client handles device discovery, writes configuration and start commands, and processes incoming sensor and battery data. It also provides a simple CLI for issuing runtime commands and saving the output locally or to an MQTT broker.

6. **Testing and Debugging**: Each component was individually tested using mocks and hardware-in-the-loop techniques. Special attention was paid to timing accuracy, state transitions, and BLE reconnection behavior. Log output was extensively used to trace state changes and identify bugs during acquisition sessions.

Each phase concluded with unit tests and partial integration tests before proceeding to the next step, ensuring traceability and regression prevention.

4.3 System Architecture

The overall architecture is composed of two tightly coupled subsystems: the **peripheral node** (Arduino Nano 33 BLE) and the **central controller** (Python-based client). Communication between them is established via Bluetooth Low Energy (BLE), with the peripheral acting as a GATT server and the central as a GATT client.

Peripheral Node (Firmware) The peripheral node is responsible for:

- Advertising its presence and accepting BLE connections.
- Receiving runtime configuration: timestamp, frequency, initial counter.
- Acquiring IMU samples at a configurable rate.
- Logging data to the SD card with robust file handling.
- Sending data via BLE notifications.
- Monitoring battery voltage and notifying the central device.
- Handling disconnection events via a fault-tolerant FSM.

All these components interact through a shared FSM engine which controls transitions and guards states based on received commands and internal events (timeouts, reconnections).

Central Controller (Python Client) The client handles the orchestration of one or more peripheral devices. Its responsibilities include:

- Scanning and discovering BLE peripherals.
- Establishing connections and writing configuration parameters.
- Subscribing to sensor data and battery characteristics.
- Handling disconnections and executing reconnection strategies.
- Buffering and storing data either as CSV or forwarding to MQTT.
- Managing user interaction through a command-line interface.

This client-centric architecture decouples acquisition logic from visualization and analysis, allowing the same peripheral firmware to be reused across different use cases (e.g., local logging vs. real-time streaming).

Peripheral Modules

- **BLE Manager**: Manages GATT server operations and event handling.
- **IMU Acquisition Module**: Periodically reads acceleration data.
- **Data Logger**: Writes structured logs to SD card.
- **FSM Controller**: Coordinates state transitions based on BLE commands and internal timers.
- **Battery Monitor**: Periodically samples and notifies battery percentage.

Central Modules (Python)

- **BLE Handler**: Scans, connects, and subscribes to notification characteristics.
- **Command Executor**: Sends timestamp, configuration, and activity commands.
- **Callback Manager**: Processes ACKs, sensor data, and battery updates.
- **Data Storage Layer**: Stores incoming data in CSV format or publishes to MQTT.

4.4 Finite State Machine

The FSM (Finite State Machine) represents the core behavioral logic of the firmware, controlling all interactions between the device, the BLE central, and the internal modules such as the IMU, logger, and battery monitor. It encapsulates the full operational lifecycle of the device, ensuring robust and deterministic transitions based on both external events (e.g., BLE commands) and internal conditions (e.g., timeouts, sensor readiness).

The FSM is structured around a set of discrete states, each responsible for a specific operational role:

- **Power On**: This is the initial state entered upon device reset or power-up. The device initializes all modules (IMU, BLE, SD card, timers), loads the configuration from the SD card, and starts BLE advertising to signal availability for connection.
- **Wait Timestamp**: Upon BLE connection, the device waits for the central to send a valid UNIX timestamp, which will serve as a temporal reference for the entire campaign. If the timestamp is malformed or missing, the device remains in this state.
- **Wait Configuration**: After receiving the timestamp, the central must send the configuration parameters, including sampling frequency and initial counter. If these parameters are invalid or missing, the device loops back waiting for a valid configuration.
- **Send ACK**: Upon successful reception and validation of configuration parameters, the device sends an acknowledgment (ACK) message back to the central. The ACK can be either positive (OK) or negative (KO), depending on the validity of the configuration.
- **Wait Start**: The device is now ready to begin the acquisition process but waits for an explicit `START` command from the central. This separation allows synchronization between multiple devices or deferred campaign initiation.
- **Acquisition**: Once the start command is received, the device begins sampling data from the IMU at the configured frequency. Each sample is simultaneously transmitted to the BLE central and stored locally on the SD card. Battery status updates and frequency changes can be handled in this state as well.
- **Stop / Sleep**: These are transitional or terminal states triggered by the central. `STOP` pauses the acquisition while maintaining state, allowing later resumption. `SLEEP` ends the current session and opens a new data file for future campaigns.
- **Disconnection / Final**: If the BLE connection is lost during any phase (especially acquisition), the device enters a temporary disconnection state and attempts to reconnect. A timeout is started. If reconnection occurs within the allowed window, the device resumes its previous state. Otherwise, it transitions to **Final**, gracefully closing files, releasing resources, and entering a halted state.

Transition Logic Transitions between states are determined by BLE write events, valid configuration payloads, user commands, and internal timers. For example:

- From `Wait Timestamp` to `Wait Configuration` upon receipt of a valid 4-byte timestamp.
- From `Wait Configuration` to `Send ACK` when the configuration packet is properly formatted.
- From `Send ACK` to `Wait Start` upon positive acknowledgment.
- From `Acquisition` to `Stop` or `Sleep` upon command 0x0F or 0xFF.
- From any state to `Disconnection` when the BLE connection drops unexpectedly.

Robustness and Error Handling The FSM is designed to tolerate communication interruptions and user misconfigurations. Invalid payloads, unknown commands, or out-of-sequence operations are ignored without affecting the system stability. Every state checks for validity before applying changes, and a global timeout mechanism ensures that disconnections are handled cleanly.

Additionally, the FSM allows for graceful resets via hardware button or software command, ensuring that the device can always return to a known stable state. The modular implementation also enables the FSM to be extended with additional states (e.g., Firmware Update, Diagnostic Mode) in future versions.

To improve readability and debugging, each transition is logged to the serial console, including timestamps and current state names. This has proven essential during development and testing phases, especially for reproducing edge cases.

4.5 BLE Communication Protocol

The Bluetooth Low Energy (BLE) protocol serves as the primary communication mechanism between the peripheral and central systems. In this project, the device is implemented as a GATT (Generic Attribute Profile) server that exposes a set of custom characteristics organized within a primary custom service. These characteristics allow configuration, control, data transfer, and monitoring of the device state.

Defined Characteristics The BLE protocol defines six main characteristics:

- **ACK (Notify)**: Sends acknowledgment messages to the central after processing configuration or control commands. A value of `0xAA` indicates success (ACK OK), while `0xFF` denotes a failure or invalid payload (ACK KO).
- **Sensor Data (Notify)**: Transmits sampled IMU data packets in binary format. Each packet contains a 16-bit counter and three 16-bit acceleration values (X, Y, Z). Notifications are sent at the configured sampling rate.
- **Timestamp (Write)**: Accepts a 4-byte UNIX timestamp sent from the central. This timestamp synchronizes the device clock with external systems and is logged to the SD card.
- **Campaign Config (Write)**: Receives acquisition configuration data. The payload may include an initial sample counter and sampling frequency or act as a confirmation for configuration preloaded from the SD card.
- **Activity Command (Write)**: Accepts 1-byte control commands to start, stop, pause, or terminate the campaign. Commands include `START` (0x00), `STOP` (0x0F), `SLEEP` (0xFF), and `END` (0xBB).
- **Battery Level (Notify/Read)**: Uses the standard BLE Battery Service UUID (180F) to expose the battery charge level. The central may read this value on request or subscribe to receive periodic updates.

Communication Workflow To ensure correct operation, the BLE central must interact with the peripheral using a strict sequence:

1. Connect to the device and subscribe to all notification characteristics (ACK, Sensor Data, Battery).
2. Write a valid 4-byte timestamp to the Timestamp characteristic.
3. Write configuration data to the Campaign Config characteristic.
4. Wait for the ACK notification (`0xAA`).
5. Send `START` to begin acquisition.
6. Optionally send frequency updates or terminate the session with `STOP`, `SLEEP`, or `END`.

Payload Format and Encoding All payloads are transmitted as raw bytes, not human-readable strings. For example, the timestamp `0x5F3759DF` is sent as four bytes in little-endian format. IMU data packets are 8-byte binary structures encoded as:

- 2 bytes: Sample counter (uint16)
- 2 bytes: Acceleration X (int16)
- 2 bytes: Acceleration Y (int16)
- 2 bytes: Acceleration Z (int16)

This compact encoding minimizes bandwidth usage and improves performance during high-frequency sampling.

4.6 Data Acquisition and Logging

Data acquisition is centered around the LSM9DS1 inertial sensor embedded in the Arduino Nano 33 BLE board. The firmware configures the IMU at startup and begins polling acceleration values once the campaign is started.

Sampling and Conversion Sampling is triggered by a timer based on the configured frequency, which is received via BLE. The `IMU.readAcceleration()` function is called at each interval, returning float values for X, Y, and Z axes in units of `g`. These values are then converted to 16-bit signed integers (scaled as needed) and packed into a binary structure.

Logging to SD Card In addition to BLE transmission, all data is saved locally to a microSD card using the SPI interface. Each session creates a new file named `DataN.txt`, where `N` is incremented per session.

- **First line**: Contains the ISO 8601 timestamp received from the BLE central.
- **Subsequent lines**: Each line logs a sample, starting with the relative timestamp in milliseconds (since boot), followed by a hexadecimal representation of the 8-byte packet.

Sample Log File Content

```
2025-07-17T15:43:12.000
2025-07-17T15:43:12.250 250 0x000503F1FFC3001A
2025-07-17T15:43:12.500 251 0x000603E9FFE2000C
2025-07-17T15:43:12.750 252 0x000703D1FFCB000F
```

Each line consists of a time offset and the corresponding measurement. This format ensures that sessions are self-contained and analyzable even without the BLE client.

Error Handling If the SD card fails to initialize, the system halts execution to prevent undefined behavior. Additionally, a failure to write (e.g., file system full) transitions the FSM to a `Final` state, ensuring safe shutdown.

Data Integrity and Traceability The use of a counter in every packet allows post-processing tools to detect dropped samples and reconstruct the full dataset. This makes the system resilient to temporary disconnections and enables reliable offline analysis.

4.7 Python Central Client

The Python central client is a critical component that orchestrates the entire BLE campaign from the user side. It is implemented as an asynchronous script using the `bleak` library, enabling non-blocking operations, responsive user interactions, and simultaneous data handling.

Core Functionalities The client carries out the following actions in sequence:

1. **Discovery and Connection**: The client scans for BLE devices advertising the target UUID and connects to the peripheral node upon detection. A timeout mechanism ensures retries in case of failures.

2. **Timestamp and Configuration Setup**: Once connected, the client writes the current UNIX timestamp (4 bytes) to the appropriate characteristic, followed by a configuration packet containing the sampling frequency and optional starting counter. This ensures synchronization across different recording devices.

3. **ACK Handling**: After sending the configuration, the client listens for the ACK response. A valid response (`0xAA`) allows the process to proceed; otherwise, the client logs the error and retries or aborts the session based on user preference.

4. **Start Acquisition**: When ACK is received, the client issues the `START` command to begin data acquisition. The client then subscribes to notifications from the Sensor Data and Battery Level characteristics to receive incoming data.

5. **Runtime Command Handling**: The client exposes a command-line interface (CLI) that allows the user to control the session in real time. Available commands include:

   - `start` — triggers acquisition.
   - `stop` — pauses the session.
   - `freq [value]` — changes the sampling frequency.
   - `shutdown` — safely terminates and closes the session.
   - `quit` — disconnects and exits without saving.

6. **Data Buffering and Storage**: Received IMU packets are buffered in memory to avoid blocking the BLE event loop. Periodically, the data is flushed to a local CSV file named with a timestamp to ensure uniqueness and reproducibility.

Reconnection and Fault Tolerance BLE connections can be fragile, especially on Linux systems with BlueZ or in noisy environments. To mitigate this, the client implements robust reconnection logic:

- On unexpected disconnection, the client waits a predefined interval (e.g., 10 seconds).
- It attempts reconnection a limited number of times.
- If reconnection succeeds, it restores the last known state and continues the session.
- If all attempts fail, the client logs the failure and closes the session gracefully.

The client also detects device restarts (e.g., due to power loss) by checking for counter resets or missing timestamps. In such cases, it alerts the user and prompts a manual reset.

Logging and Debugging To aid development and field testing, the client generates structured log files containing:

- Connection and disconnection events.
- Timestamps of sent and received packets.
- ACK status and configuration feedback.
- Error messages and BLE stack traces.

Logs are color-coded in the terminal and saved in plain text format for later analysis. This proved critical during the validation of edge cases and timing-related bugs.

Extensibility The client is modular, allowing easy extension to support new commands, multi-device campaigns, or data forwarding to cloud platforms. A planned future enhancement includes MQTT streaming for real-time dashboards and remote control.

4.8 Challenges Faced

- **BLE Instability**: Some BLE operations hang due to library limitations (especially on Linux with BlueZ).
- **IMU Reset Detection**: After a device reset, internal counters may misalign. Solution: hard fail if no data is received.
- **Cross-platform BLE APIs**: Different OSes require specific backends (WinRT, CoreBluetooth, BlueZ).
- **Precise Timing**: Sampling accuracy depends on compensation factor due to OS delays.
- **SD Logging Race Conditions**: Avoid writing during state transitions.
- **Debugging Asynchronous Code**: Requires structured logging and time-stamped output.

Despite these difficulties, the system was successfully stabilized and thoroughly tested under multiple scenarios, including multiple acquisition campaigns, disconnection recovery, battery drain, and IMU noise bursts.

5 Technical Documentation

5.1 Data Schemas

The system uses three primary data formats during its operation: BLE binary packets, SD card log files, and optional CSV output on the central side. Each is designed for compactness, readability, and ease of post-processing.

1. BLE Binary Data Packet (Sensor Data Notification) Each notification packet sent by the peripheral to the central contains raw accelerometer data encoded in binary format:

- 2 bytes: Sample Counter (uint16, little-endian)
- 2 bytes: Acceleration X (int16, little-endian)
- 2 bytes: Acceleration Y (int16, little-endian)
- 2 bytes: Acceleration Z (int16, little-endian)

Total size: 8 bytes.

2. SD Card File Format (`DataN.txt`) Data collected during a session is stored on the SD card with the following structure:

- First Line: ISO 8601 timestamp received from the central.
- Subsequent Lines: `0x` where:
  - is the time in milliseconds since boot.
  - is the hexadecimal encoding of the 8-byte packet.

Example:

```
2025-07-17T15:43:12.000
250 0x000503F1FFC3001A
500 0x000603E9FFE2000C
750 0x000703D1FFCB000F
```

3. Central CSV Format When the BLE central stores data locally, it converts the binary packets into human-readable CSV rows with the following columns:

- Timestamp (ISO 8601)
- Sample Counter
- Acceleration X (g)
- Acceleration Y (g)
- Acceleration Z (g)

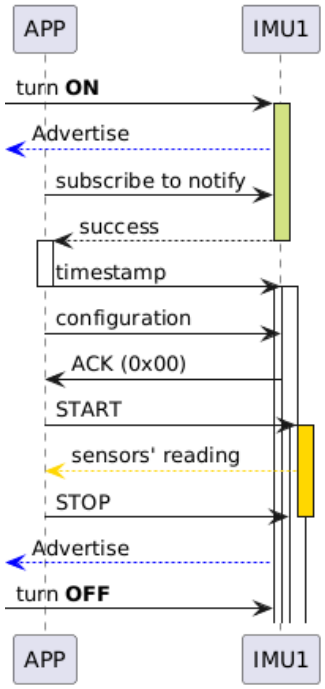
Example:

```
2025-07-17T15:43:12.250, 5, 0.992, -0.876, 0.041
2025-07-17T15:43:12.500, 6, 0.985, -0.901, 0.022
2025-07-17T15:43:12.750, 7, 0.976, -0.923, 0.030
```

This schema ensures full data traceability and interoperability with analysis tools such as Python (pandas), Excel, or MATLAB.

5.2 Interaction Flow (Sequence Diagram)

The system architecture can be represented by a two-layer block diagram, separating the embedded BLE device and the central Python client.



Description

- **turn ON**: The IMU device boots and begins BLE advertising.
- **Advertise**: The device becomes discoverable to the client.
- **Subscribe to notify**: The APP subscribes to all required characteristics (e.g., ACK, Sensor Data).
- **Timestamp and Configuration**: The APP writes a UNIX timestamp and campaign parameters (sampling rate, counter).
- **ACK**: The IMU responds with a 0x00 ACK if configuration is valid.
- **START**: The APP sends the start command.
- **Sensors' Reading**: IMU1 begins sampling and sends data via notifications at the configured rate.
- **STOP**: The session is stopped by the client.
- **Advertise**: The device returns to advertising state, ready for a new session.
- **turn OFF**: The device is manually powered down or enters sleep mode.

This flow ensures a deterministic and synchronized interaction between the components. It reflects the logic implemented in both the firmware Finite State Machine (FSM) and the Python client BLE handler.

6 Conclusions

6.1 Summary

This project presented the complete redesign and implementation of an IoT data logger device using Bluetooth Low Energy (BLE) and an embedded IMU. The system replaces a legacy ANT-based architecture with a modular, robust BLE solution capable of real-time data acquisition, local storage, and runtime configuration. The firmware is driven by a Finite State Machine (FSM) that governs all operational states, from connection and configuration to data logging and disconnection recovery. The central Python client complements the peripheral node by handling BLE communication, user interaction, and data persistence.

The main achievements of the project include:

- A custom BLE GATT protocol tailored for sensor campaigns.
- Reliable SD card logging of raw IMU data in hexadecimal format.
- A cross-platform, asynchronous Python BLE client with runtime CLI.
- Reconnection and error handling mechanisms.
- Timestamp-based synchronization and battery monitoring.

6.2 Lessons Learned

Throughout this project, several key technical and practical insights were gained:

- **BLE Design Considerations**: BLE's asynchronous nature and limited bandwidth required careful planning of packet size, frequency, and timing to ensure reliability.
- **State Management in Embedded Systems**: Using a Finite State Machine helped structure the firmware logic, isolate bugs, and recover from failures more effectively.
- **Python Async Programming**: Developing an asynchronous client in Python using `asyncio` and `bleak` provided practical experience with event-driven architectures.
- **Cross-platform Challenges**: Ensuring BLE compatibility across Linux, macOS, and Windows introduced platform-specific issues that were addressed using abstraction layers.
- **Testing Without GUI**: Designing without a user interface emphasized the importance of clear logging, structured outputs, and a predictable CLI for usability.

6.3 Future Work

While the core functionality is complete, the system provides a solid foundation for several future enhancements:

- **OTA Firmware Updates**: Add over-the-air update capability to simplify deployment and versioning.
- **Extended Sensor Support**: Expand the firmware to support other sensor types (e.g., gyroscope, magnetometer, GPS).
- **Power Optimization**: Explore deep sleep modes and smarter sampling to reduce energy consumption during long campaigns.

This work lays the groundwork for a modular, extensible BLE-based sensor platform suitable for research, education, and prototyping in motion tracking and IoT.