Development of an Internet of Things Gateway for Interfacing with Bluetooth Low Energy Peripherals

by

Joshua Porter

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science in Engineering

in

Electrical Engineering

YOUNGSTOWN STATE UNIVERSITY

May 2025

Development of an Internet of Things Gateway for Interfacing with Bluetooth Low Energy Peripherals

Joshua Porter

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

_____

Joshua Porter, Student Date

Approvals:

_____

Vamsi Borra, PhD, Thesis Advisor Date

_____

Frank X. Li, PhD, Committee Member Date

_____

Ghassan Salim, Committee Member Date

_____

Severine Van slambrouck, PhD, Graduate Studies Date

ABSTRACT

Internet of Things (IoT) devices, such as thermostats, lighting systems, and fitness trackers, have revolutionized both residential and industrial environments, enabling users to remotely control and manage them. Although many IoT devices are often manageable through Original Equipment Manufacturer (OEM) software applications, overseeing devices from various OEM origins simultaneously, or even customized hardware, can be complex and tedious. To address this challenge, an IoT gateway serves as a centralized hub that supports wireless connectivity across various protocols. Bluetooth Low Energy (BLE), a widely adopted wireless communication protocol in low-power-consuming devices such as sensors, is therefore employed in many IoT gateways. Large-scale IoT networks significantly benefit from an IoT gateway, as it provides a unified management point for all connected devices. OEMs of IoT gateways may offer a software development kit (SDK) to facilitate application customization, enabling the attainment of specific design requirements. This thesis presents the design, development, and implementation of two software applications to acquire real-time sensor data from custom BLE-enabled printed circuit boards (PCBs). Leveraging an IoT gateway, its compatible SDK, and a library of sample programs, two applications are developed to monitor sensor data: a serial terminal interface and a dynamic web-based dashboard.

ACKNOWLEDGMENTS

First, I would like to thank my thesis advisor, Dr. Vamsi Borra, and the director of the Rayen School of Engineering at Youngstown State University (YSU), Dr. Frank Li, for appointing me as a graduate assistant in the department of electrical and computer engineering at YSU to support my thesis research and for their invaluable guidance.

Next, I would like to thank my numerous teachers and professors throughout my education, especially Mr. Ghassan Salim. Their enthusiasm, expertise, and commitment to helping students succeed have left a lasting impact on my life. Without these select few, I would not have developed a significant enthusiasm for learning or pursued the daunting challenge of completing a master's thesis.

Next, I would like to thank my fellow graduate assistants, who brightened each day in the lab, provided knowledgeable assistance, and offered meaningful words of encouragement. Additionally, I would like to thank my undergraduate lab students, who welcomed me as a teaching assistant, enabling me to strengthen both my teaching skills and expertise.

Finally, I would like to thank my family and friends for their enduring love and support. Their invaluable guidance has encouraged me to keep striving for my life goals, and I could not be more grateful for that.

This thesis is dedicated to my grandmother, who regrettably passed away during its writing. Although she could not see its completion, she always believed I could achieve anything I set my mind to.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API – Application Programming Interface

BLE – Bluetooth Low Energy

CCS – Code Composer Studio

CSS – Cascading Style Sheets

DAQ – Data Acquisition

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

IIoT – Industrial Internet of Things

IoT – Internet of Things

JSON – JavaScript Object Notation

JS – JavaScript

JTAG – Joint Test Action Group

MAC – Media Access Control

MCU – Microcontroller Unit

OEM – Original Equipment Manufacturer

PCB – Printed Circuit Board

SIG – Special Interest Group

SDK – Software Development Kit

TI – Texas Instruments

UI – User Interface

URI – Uniform Resource Identifier

USB – Universal Serial Bus

UUID – Universally Unique Identifier

YSU – Youngstown State University

CHAPTER 1 – INTRODUCTION

The Internet has become a fundamental component of modern infrastructure, revolutionizing communication, information exchange, and business operations. Devices like computers, smartphones, and wearables are prime examples of how the Internet has permeated modern society. These devices, along with others, comprise the Internet of Things (IoT), a broad term encompassing interconnected devices that collect, control, analyze, and share data in real-time over the Internet [1]. In 2024, industry leaders estimated a worldwide total of 17.9 billion IoT-connected devices, with this number expected to have doubled by 2032 [2]. This graphical forecast is displayed in Figure 1.

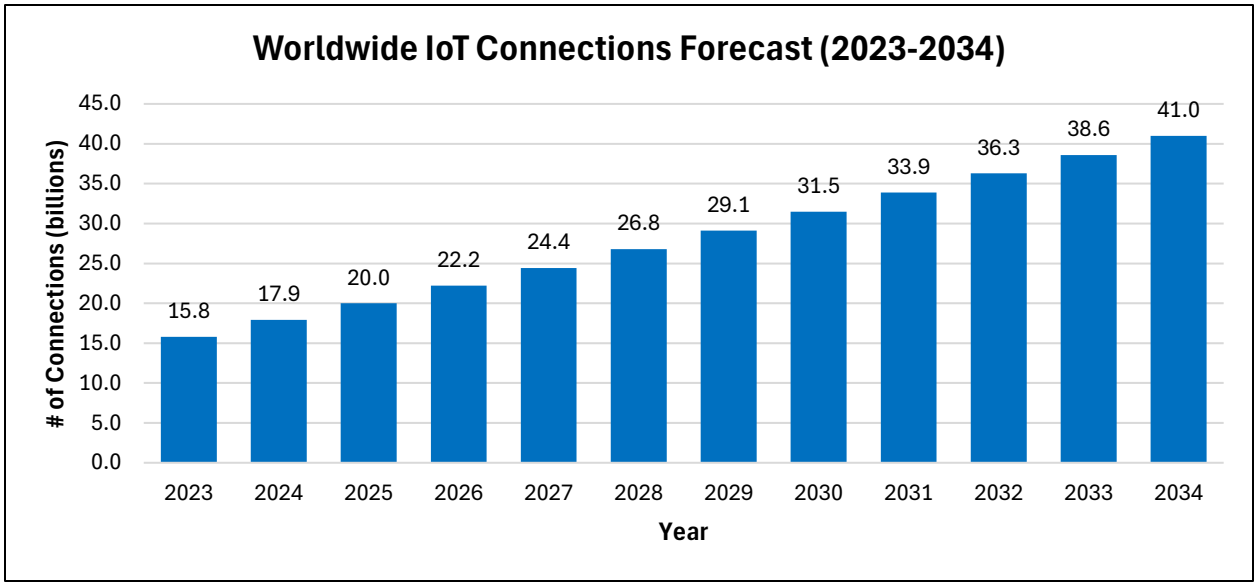

Figure 1: Worldwide IoT connections forecast (2023-2034) [2].

For instance, low-power-consuming devices, such as sensors and antennas [3], [4], [5], utilize communication protocols optimized for low power consumption. Bluetooth Low Energy (BLE) is a widespread protocol that facilitates communication between low-power peripheral devices and high-power central devices [6]. Unlike the need for special controllers for high-energy applications and heavy industrial equipment [7], [8], end devices in the day-to-day are primarily operated on

low power. BLE centrals, unlike BLE peripherals, typically establish Internet connections; therefore, when a peripheral connects and transfers data to a central, the central handles the peripheral's data transfer to the Internet, thus completing a local IoT network. Bluetooth is a widely used communication standard in many IoT networks and can be further classified into two protocols: Bluetooth Classic and BLE. Evidently, BLE consumes less energy than Bluetooth Classic, making it a more sensible choice for integrating into technology with low-power requirements, such as sensors and wearables.

BLE stands out from other low-power wireless communication protocols such as Zigbee due to its high data rate, low power consumption, and seamless integration into modern smartphones. Even though Zigbee is widely used in low-power applications and is more reliable than BLE for large IoT networks, it has a significantly lower data rate, and it requires a centralized hub with the Zigbee protocol [1]. Therefore, BLE will suffice for the design outlined in this thesis, as the working IoT network consists of only a handful of devices integrated with the BLE protocol.

The rest of this chapter will provide background information on emerging IoT applications and a summary of Bluetooth's history. Subsequently, the significance of the research and its objectives will be presented, followed by a structural outline of this thesis.

1.1 Background Information

Before examining the research presented in this thesis, it is essential to present current and emerging IoT applications to emphasize their relevance in modern society. Additionally, a brief history of Bluetooth technology will highlight its development and discuss its role in supporting many of these IoT applications.

1.1.1 Emerging IoT Applications

Industries are utilizing IoT devices to automate tasks that once required manual labor. In turn, they can expect more efficient output and more accurate data recordings, as humans cannot compete with the speed and precision of a computer. Actions that once required considerable time are now nearly instantaneous as users acquire data from wirelessly connected devices, such as computers or smartphones. Swamy and Kota [1] describe the domains within their defined ten emerging IoT applications, displayed in Figure 2.

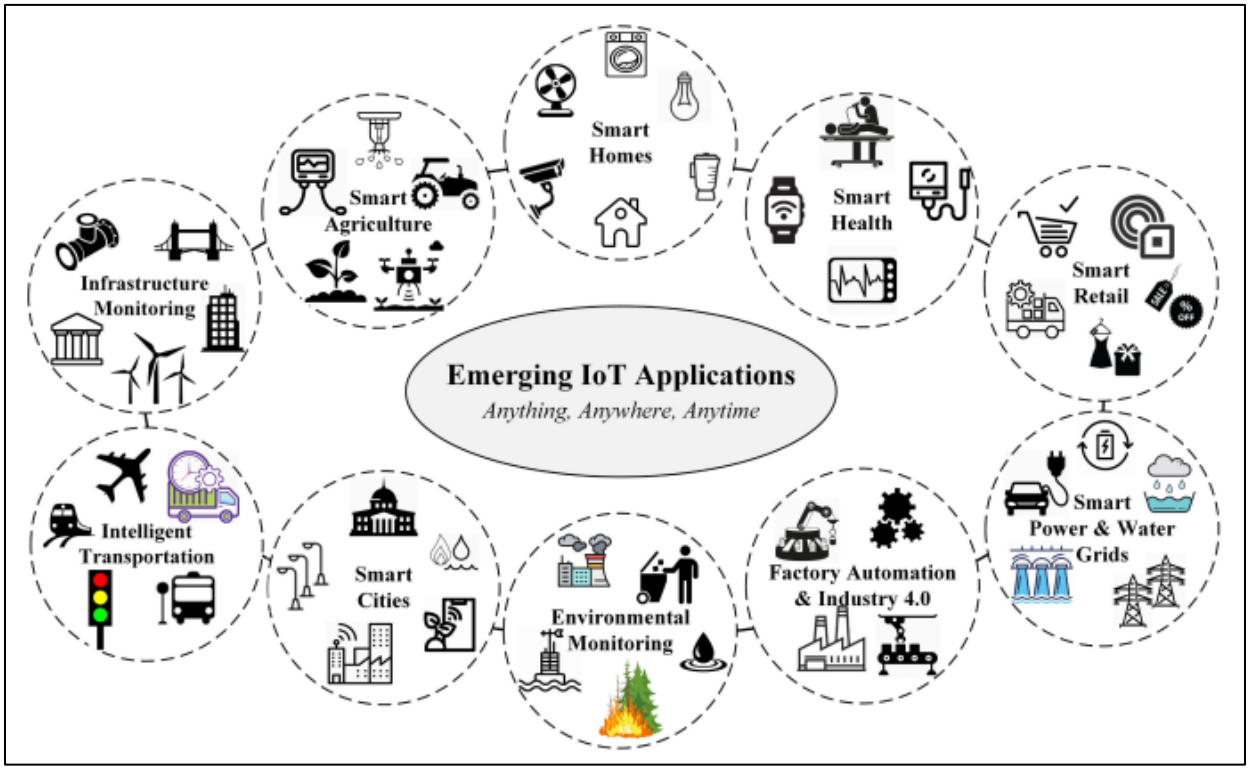

Figure 2: Emerging IoT applications [1].

The following are concise examinations of all ten applications, starting with Infrastructure Monitoring and moving clockwise with respect to the layout of Figure 2:

- Infrastructure Monitoring: Civil structures with sensors efficiently scan for damage, reducing ecological, financial, and humanitarian risks. Intelligent military surveillance systems swiftly and precisely detect threats and rescue victims using advanced

identification systems. Data acquired manually in the infrastructure industry is not cost-efficient; therefore, deploying IoT devices will save companies valuable resources.

- Smart Agriculture: Deep learning models detect crop diseases and weeds at initial stages, smart greenhouses maintain optimal conditions, and livestock behavior and health are monitored from afar. Most farmers still rely on traditional farming techniques. However, integrating IoT devices helps mitigate traditional farming issues, save resources, and increase crop yield.
- Smart Homes: Appliances are automated with a controlling device, such as a smartphone. Air quality is monitored to alert residents to pollutants, garden plant nutrition is observed and maintained remotely, intelligent surveillance quickly identifies threats near the property, and elderly residents at risk of fatal diseases can monitor their health remotely. Homeowners who utilize IoT devices experience an increased quality of life, as manual intervention is no longer required, and security is maintained.
- Smart Health: Diseases and disorders are diagnosed early with integrated sensors. Remote real-time health monitoring systems for hospital patients are available. Smart devices track calories burned, sleep cycles, and heart rate, and emergency care operators quickly locate nearby hospitals. The presence of IoT devices in the healthcare industry helps prevent mistakes that could lead to further injury or death.
- Smart Retail: Sensors enable customers to locate items quickly, Radio frequency identification tags assist store owners in managing inventory, self-checkouts are automated to scan barcodes using cameras, and virtual reality (VR) will soon facilitate remote shopping. With enhanced algorithms to personalize the customers' shopping experience, retailers will benefit from IoT technology.

- Smart Power and Water Grids: Power grids have power loss monitoring and load balancing systems. Water grids monitor parameters such as water flow, pressure, and quality. Management systems for power and water monitor electrical energy and water consumption, respectively. With IoT devices, power and water grids maximize efficiency.
- Factory Automation and Industry 4.0: Industry 4.0, the latest stage of the industrial revolution, leverages IoT technology to enhance manufacturing efficiency. Intelligent surveillance predicts and analyzes injuries in the workplace, while supply management operations are simplified through real-time tracking of products. Additionally, smart quality control operations analyze products in the final production stage.
- Environmental Monitoring: IoT technology deployed in urban areas enhances residents' quality of life. Detection models identify forest fires at early stages, water distribution and sanitization services are optimized with digital metering systems, outside air quality is continuously monitored, and alert systems alert residents of natural disasters.
- Smart Cities: Smart cities are designed to improve societal well-being by providing IoT services to city officials and residents. Smart infrastructure systems optimize resource allocation, e-governance systems enable faster decision-making in government organizations and transparency in governing agencies, surveillance systems increase public safety by detecting and preventing crime, and urban residents control energy, water, and waste management.
- Intelligent Transportation: Transportation forms, including roads, air, sea, and rail, are improved with integrated IoT technology. Cameras in vehicle-dense areas constantly control and monitor traffic, providing travelers with accurate traffic updates. Traveler

information systems provide travelers with live updates along the desired route, and vehicle-to-vehicle communication offers information about surrounding vehicles and infrastructure in autonomous vehicles [1].

Swamy and Kota's report on emerging IoT technologies [1] supports the projected growth of worldwide IoT connections over the next decade, as shown in Figure 1 [2]. Many industries, including agriculture, manufacturing, and infrastructure, are utilizing this technology to help maximize product output, mitigate hazards, and increase societal well-being [1]. Therefore, the importance of IoT technologies in present times is evident, as well as their continued growth in usage in the years to come.

1.1.2 Bluetooth Overview

The foundation of Bluetooth dates to 1994, when Dr. Jaap Haartsen, a Dutch electrical engineer working for Ericsson in the United States at the time, devised a system that would enable wireless connectivity between electronic devices. Initially intended as a convenient alternative to wired voice calls, it eventually expanded to support a broader range of devices, including computers, smartphones, microphones, speakers, and more. Haartsen admitted, "To be honest, I did not have any idea of how big Bluetooth could become." Haartsen later played a critical role in the founding of the Bluetooth Special Interest Group (SIG) in 1998 [9], a non-profit international standards development organization comprising over 40,000 companies, which oversees the regulations, licensing, and development of Bluetooth technology [10], [11].

While Bluetooth technology grew in popularity, a concern was its power demand and the desire to integrate Bluetooth into smaller, low-power-consuming devices [12]. In 2001, Nokia started working on its wireless connectivity technology called Wibree, a lower power-

consuming alternative to Bluetooth, also referred to as Bluetooth Classic [13], but operating on the same 2.4 GHz frequency band [14]. Nokia officially launched Wibree five years later, in 2006 [12], and the Bluetooth SIG, seeing an opportunity, later absorbed Wibree in 2010 in Bluetooth Core Specification 4.0, renaming it to Bluetooth Low Energy [13], [15].

In 2023, the Bluetooth SIG reported an estimated 5 billion Bluetooth-enabled devices shipped worldwide, forecasted to grow to 7.5 billion worldwide shipments in 2028, a projected eight percent compound annual growth rate over five years [16]. Three decades after its inception, Bluetooth remains a market leader in wireless connectivity, and this trend is expected to continue for years to come.

Today, Bluetooth technology, specifically BLE, plays a pivotal role in the IoT. BLE is widely integrated into modern smartphones, enabling seamless connectivity with other BLE-enabled devices such as thermostats, lighting systems, and fitness trackers. Original Equipment Manufacturers (OEMs) often provide software applications that facilitate real-time control and monitoring, offering user-friendly solutions for consumers.

1.2 Research Overview

1.2.1 Significance

Each IoT network is unique, varying in complexity, size, and protocols. A residential network, for example, may consist of only a handful of devices, whereas an industrial network may comprise hundreds of devices. Regardless of network scale or protocol diversity, many IoT environments can significantly benefit from an IoT gateway, a central hub that supports numerous protocols and concurrent connections. IoT gateways can vary in scalability, robustness, security, and protocol support, so users must select the correct gateway to meet their network's requirements [17], [18]. OEMs of IoT gateways may provide a compatible Software

Development Kit (SDK) to customize their applications, allowing developers to meet specific design requirements.

This thesis presents the design, development, and implementation of a unique IoT network, where an IoT gateway interfaces with customized printed circuit boards (PCBs) integrated with sensors via BLE, enabling tailored data acquisition (DAQ) applications. The system presented in this thesis is an addition to previous work conducted by Yarwood, Garretto, and Kuzior [19], [20], [21]. This included designing the BLE-enabled PCB, developing its sensor firmware, and implementing a custom smartphone DAQ application. However, the smartphone application has limitations: it can only connect to one PCB at a time due to Bluetooth hardware limitations and requires constant proximity between the smartphone and the PCB. In our previous study, we demonstrated that it is possible to detect the presence of Bovine Serum Albumin (BSA) at varying concentrations through a portable and cost-effective BLE-enabled device [22]. The newly proposed IoT gateway addresses these limitations by supporting at least four concurrent BLE connections and providing broader wireless access for users.

1.2.2 Objectives

The primary objective of this research is to develop embedded software applications for an IoT gateway that enable real-time sensor DAQ from custom BLE-enabled PCBs. Two applications will be created using the gateway's SDK: a serial terminal program for convenient local monitoring, and a dynamic website for remote access from virtually any Internet-accessible device. This dual-interface approach significantly improves the previous system, which supported DAQ from only one PCB at a time.

To accomplish the primary objective, several sub-objectives must first be addressed:

1. Select an IoT gateway that meets the design requirements.
2. Analyze the sensor's functionality and the BLE data transmission structure of the custom PCBs.
3. Identify a suitable development starting point within the IoT gateway's SDK framework.
4. Develop the website interface after confirming the successful implementation of the terminal application.

1.3 Thesis Structure

This thesis is organized into four additional chapters following this first introductory chapter. Chapter 2 provides a comprehensive review of related post-graduate research, highlighting the current state of technologies and addressing existing knowledge gaps. Chapter 3 outlines the methodology employed in this study, beginning with a general overview and progressing to technical specifics. Chapter 4 presents the research results along with a detailed discussion and analysis. Lastly, Chapter 5 delivers concluding remarks and proposes prospective additions and improvements to the system.

CHAPTER 2 – LITERATURE REVIEW

This chapter reviews published works with applications pertinent to this thesis. Each discussed paper will be thoroughly examined and concisely presented, stating its purpose, key findings, and relevance to this paper. Ultimately, this chapter aims to highlight relevant applications, draw similarities between the research described in this paper and existing literature, and emphasize the originality and benefits of the research conducted for this thesis.

2.1 IoT Gateway Solutions

IoT gateways enable seamless communication across diverse networks by supporting various communication protocols. Unlike traditional network gateways that only handle Internet service, IoT gateways provide multi-protocol support, including Wi-Fi, Ethernet, dual-mode Bluetooth (Bluetooth Classic and BLE), Zigbee, LoRaWAN, and more. This ability to facilitate connections with various communication protocols makes them essential for IoT deployments. When interfacing with an extensive network of devices, IoT gateways are especially beneficial since they can translate and display large data streams in real-time for user management and monitoring through a custom web browser page or a cloud platform like Amazon Web Services (AWS) or Microsoft Azure.

One practical application of an IoT gateway can be seen in the work of Glória et al. [23], who developed a Raspberry Pi, a small computer that features Internet connectivity, as an IoT gateway for real-time monitoring and control of a swimming pool. Although this is a simulated prototype design, it includes various sensors for measuring the water's temperature and humidity, water level, a light detector, and a motor driver for water circulation. The Raspberry Pi sends this information to be viewed in a web browser via the Message Queueing Telemetry Transport (MQTT) protocol, a widely used data transmission protocol within the IoT due to its low

bandwidth consumption. Although this design was successful, many physical connections were used, which is unsafe for real-life swimming pool environments. Therefore, this design could be improved with waterproof wireless sensors that use a low-power communication protocol like BLE, Zigbee, or LoRaWAN.

While consumers enjoy IoT technologies at home, industries utilize IoT technologies to create a safe and efficient working environment. IoT technologies implemented in industrial settings are categorized within the Industrial Internet of Things (IIoT) [1], [24]. Liu et al. [24] investigate the implementation of IIoT technologies in a cloud manufacturing system with an IIoT gateway. A 3-D printer and a computer numerical control (CNC) machine were evaluated for feasibility and the advantages of the approach, with a Raspberry Pi as the acting IIoT gateway for both machines. Both case studies concluded that the open-source cloud-based solutions efficiently manage real-time data and decisions. However, the studies rely too much on open-source software rather than developing customized software. Adding tailored functions to perform specific tasks could enhance the overall user experience.

Lojka et al. [25] proposed an IIoT gateway architecture that integrates sensor networks with Human-Machine Interfaces (HMI) and Manufacturing Execution Systems (MES) to enhance remote monitoring and control. Their solution utilizes Service-Oriented Architecture (SOA), context-based services, and machine learning (ML) techniques to manage data flow and reduce human intervention. The gateway dynamically groups sensors based on contextual parameters such as temperature and location, enabling efficient event detection and query processing through a publish and subscribe model. Although the simulation validated the system, it lays the groundwork for intelligent, scalable, and flexible DAQ in industrial environments. The proposed architecture aligns with Industry 4.0 objectives by improving communication between operational

and business layers while enabling real-time decision-making at the network's edge. While the focus of their work is relevant for industrial settings, the following steps would involve deploying the gateway with real hardware to evaluate performance and reliability in live industrial settings, which is a focal point that this thesis will aim to cover.

Altogether, these studies showcased the effectiveness of IoT gateways in enabling data transmission across the Internet. A recurring pattern in several works is using a Raspberry Pi to act as an IoT gateway. However, this thesis will adopt a different IoT gateway to be integrated into a custom network, expanding the range of implementation strategies. Additionally, some studies rely heavily on open-source software for DAQ, something this thesis will address by embedding custom-developed applications into the IoT gateway. By diverging from commonly used methods for DAQ, this research will provide new insights into the full potential of custom software applications for real-time sensor monitoring.

2.2 BLE – Synopsis and Applications

Traditional methods of communication rely on wires to send and receive data. Ethernet, for example, has been a staple in Internet communication for several decades and is more stable and secure than its wireless counterpart, Wi-Fi. Due to the consumer demands of fast data transfer rates, the tradeoff is greater power consumption and a higher probability of wireless network instability. Meanwhile, wireless communication protocols such as BLE, Zigbee, and LoRaWAN do not transmit at high data rates and are consequently low-power consuming and sufficiently stable. These protocols and more are integral in the continuously evolving IoT landscape, which includes sensors and wearable technology powered by low-power coin cell batteries.

This thesis involves interfacing with BLE-enabled devices, making it essential to provide a detailed background about the BLE protocol. The first subsection will examine BLE’s theory of

operation, including its layered software architecture called the BLE stack. The second subsection will highlight custom BLE applications in scholarly writing to support BLE's importance and developments within the IoT.

2.2.1 BLE Stack

The BLE protocol is commonly visualized as a layered structure called the BLE stack. As seen in Figure 3, there are three distinct layers: the application layer (APP), the host, and the controller [26]. This section will briefly discuss each layer and its sub-layers, if applicable.

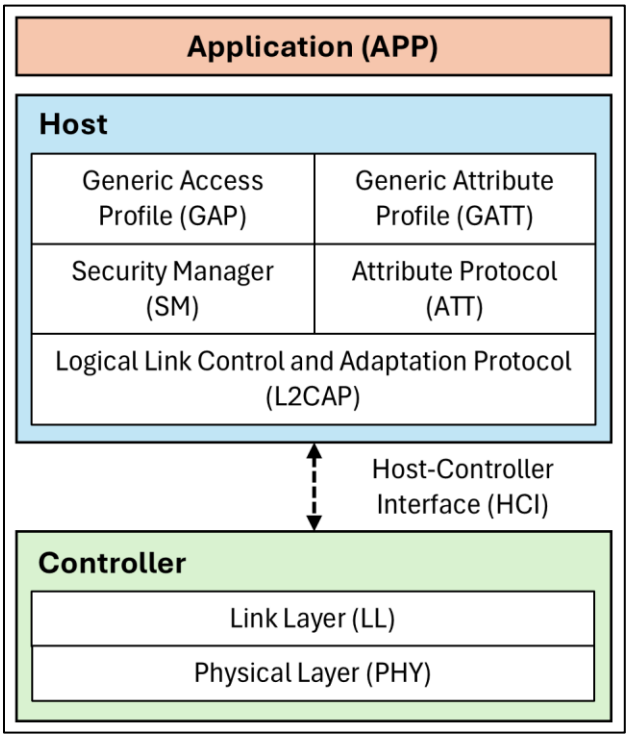

Figure 3: BLE stack architecture.

The controller at the bottom of the BLE stack consists of two sublayers: the physical layer (PHY) and the link layer (LL) [26]. The PHY represents the 2.4 GHz radio that the BLE device uses to wirelessly send and receive data. Specifically, the radio operates within the 2.4 to 2.4835 GHz frequency range, with 40 active channels spaced apart by 2 MHz. Three channels: 37, 38, and 39, are reserved for device advertising, while the other channels are used for data packet

exchange between devices [6], [26]. The LL directly interfaces with the PHY and defines the type of communications that can be created between BLE devices by managing the radio link state. Additionally, LL defines four device roles: master, slave, advertiser, and scanner [26].

The host-controller interface (HCI) does not exist within the host or the controller, however it is a crucial link between both layers. A set of rules exists within the HCI for translating raw data into packets, sending them via serial communication to the host from the controller, or vice versa [26].

The host in the middle of the BLE stack is entirely software-based, and it is comprised of five sublayers: the logical link control and adaptation protocol (L2CAP), the security manager (SM), the attribute protocol (ATT), the generic attribute profile (GATT), and the generic access profile (GAP). The L2CAP handles data from the LL and multiplexes it for the ATT and SM in a process called recombination or vice versa, which is called fragmentation [26]. The SM utilizes security algorithms for encrypting and decrypting data packets, which ensure secure connections, prevent man-in-the-middle attacks (MITM), and reduce power consumption during the connection process.

The ATT defines the communication roles of client and server, in which the client is the device that requests data from the server [6], [26]. Upon receiving the request, the server sends data back to the client. Additionally, the ATT organizes data into attributes. Each attribute is assigned a handle, a Universally Unique Identifier (UUID), a set of permissions, and a value [26]. Using the ATT, GATT establishes the framework for organizing and exchanging data on a BLE server. This data is organized into hierarchical structures called services, which group related characteristics together characteristics. Each characteristic includes a value and may be accompanied by descriptors that provide additional metadata or configuration. Access control,

such as reading and writing permissions, is applied in the ATT. The Bluetooth SIG defines standard services and characteristics using 16-bit UUIDs, while developers can define custom ones using 128-bit UUIDs. Fundamentally, services, characteristics, and descriptors are all implemented as attributes within the ATT [26].

The GAP specifies modes and procedures for device and service discovery, as well as managing connection establishment and security. Additionally, the GAP defines four device roles: peripheral, central, broadcaster, and observer. A peripheral, typically a device with low power consumption, establishes a connection to a central. However, centrals are generally high-power-consuming and can establish concurrent connections to multiple peripherals. Broadcasters, similar to peripherals, and observers, similar to centrals, do not establish connections. Consequently, all the broadcasters' data must be sent through their BLE advertisements, in which case the observer can receive these advertisements [6].

The APP at the top of the BLE stack represents the layer seen from the user's perspective. Application profiles defined by the Bluetooth SIG are placed in the APP for simple interoperability across devices from different OEMs. Additionally, the Bluetooth specification permits the creation of custom profiles for specialized usages not defined by the SIG [26].

In previous work, a custom profile was created to organize the custom services that the custom BLE-enabled PCB hosts. Its attributes will be examined later in this thesis, which hold characteristics tied to each of its sensors. These BLE characteristics are essential links for developing custom applications.

2.2.2 BLE Applications

Due to its low power consumption and integration into modern smartphones, BLE has numerous applications ranging from healthcare and sports medicine to home integration and

environmental monitoring. De Fazio et al. [27] present a wearable chest band respiratory monitor to aid those with respiratory issues. This chest band utilizes just one sensor: a custom piezoresistive strain sensor (EeonTex LTT-SLPA-20K). Data is processed efficiently using various filtering techniques and is sent to a microcontroller unit (MCU) to facilitate BLE communication. A mobile application was also created for real-time DAQ from the EeonTex LTT-SLPA-20K. Alfian et al. [28] propose a BLE-enabled personalized healthcare monitoring system for diabetic patients. This includes utilizing commercial sensors, processing real-time data through Apache Kafka, storing data on MongoDB, and integrating machine learning techniques. Hughes et al. [29] investigate the development of a BLE sensor network for identifying construction noise and sound locating. The authors also test and compare wireless nodes that use different protocols to evaluate power consumption, range, data rate, etc. These developments are critical to understanding the current state of Bluetooth technology, more specifically, BLE.

CHAPTER 3 – METHODOLOGY

This chapter presents an overview of the IoT network's requirements and overall architecture, followed by detailed descriptions of its components' functional roles and development processes.

3.1 Network Requirements and Architecture

The initial requirement of this design was to locate an IoT gateway that would support at least four concurrent BLE connections to the YSU Tags, provide a compatible SDK for development, offer serial port communication for the serial terminal application, and feature an embedded Hypertext Transfer Protocol (HTTP) server for the dynamic website. The xPico 250 Evaluation Kit from Lantronix Inc. [30] satisfied all these design requirements.

In previous work, four identical PCBs, designated as "YSU Tag," were manufactured. The device's name is derived from its manufacturing origin at Youngstown State University (YSU), with "Tag" indicating that it incorporates sensors. Each YSU Tag was given a letter label between A and D. Therefore, they will be referred to as "YSU Tag A," "YSU Tag B," "YSU Tag C," and "YSU Tag D."

The proposed IoT network architectures for the terminal and website applications are shown in Figures 4 and 5, respectively. Both networks integrate the xPico 250 into the design to enable concurrent BLE connectivity for up to four YSU Tags. "Concurrent" implies that the BLE central (xPico 250) appears to connect to all the BLE peripherals (YSU Tags) simultaneously. In theory, however, the BLE central scans for BLE peripherals' Radio Frequency (RF) advertisements [6], establishes a connection to one peripheral, and periodically switches connections to other peripherals.

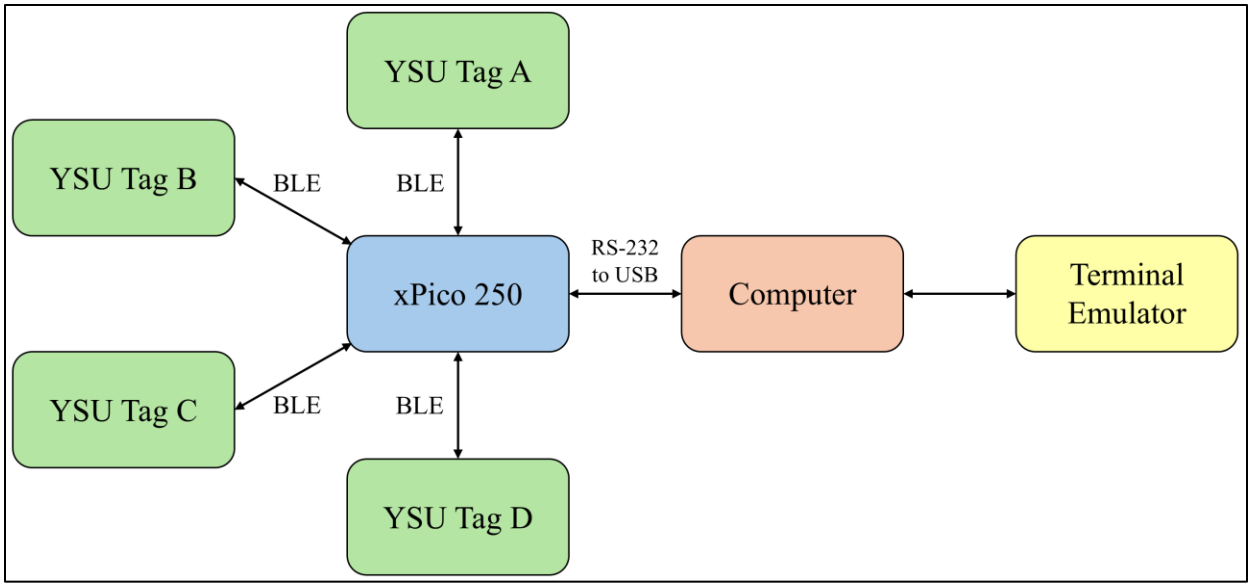

Figure 4: IoT network architecture – Terminal application.

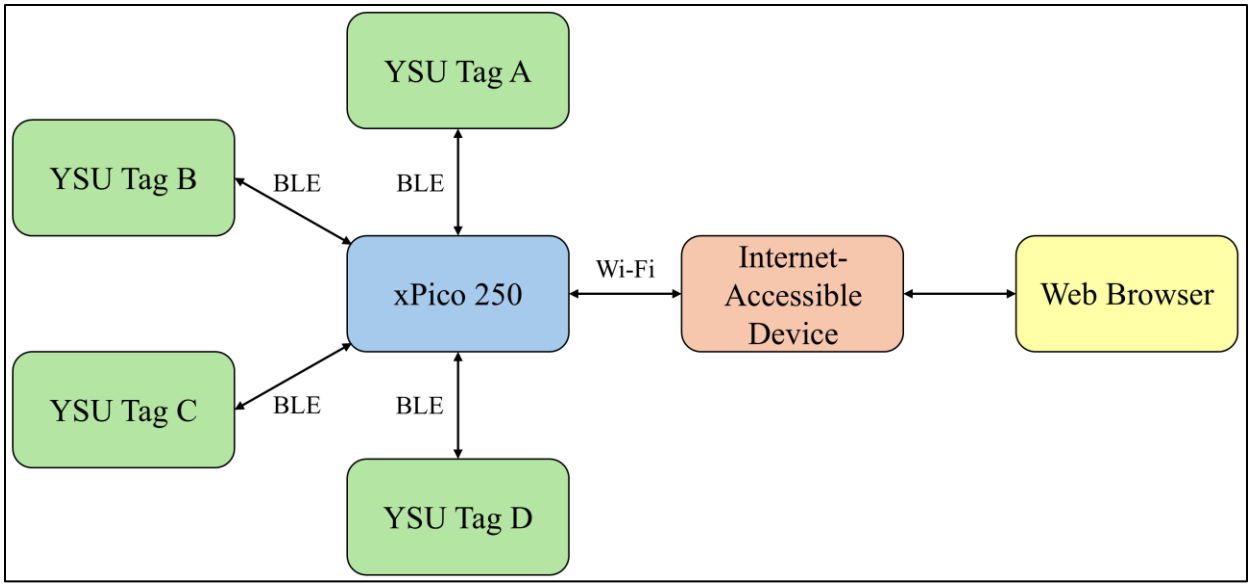

Figure 5: IoT network architecture – Website application.

The primary distinction between the two networks is how the xPico 250 interfaces with each application. Figure 4 shows that the terminal application utilizes wired communication via an RS-232 to Universal Serial Bus (USB) cable for serial data transmission. In contrast, Figure 5 shows that the website application uses wireless communication via the xPico 250's Wi-Fi access point. Additionally, the terminal application can only be accessed from a terminal emulator on a

computer, whereas the website can be accessed from virtually any Internet-accessible device. Ultimately, both applications accomplish similar tasks using different methods, resulting in distinct visual outputs.

3.2 BLE Peripheral – YSU Tag

The customized BLE-enabled PCB, "YSU Tag", shown in Figure 6, is integrated with three distinct sensors. First is Bosch's BME688: a four-in-one environmental sensor that measures temperature, relative humidity, barometric pressure, and gas resistance. The other two sensors, manufactured by Analog Devices, are the ADXL343, a 3-axis accelerometer that measures acceleration along each defined axis, and the AD5941, an electrical impedance sensor that measures an external load's impedance in polar form: magnitude and phase. Figure 6 also displays a CR2032 3V coin cell battery capable of powering the YSU Tag when inserted into the battery placeholder on its reverse side.

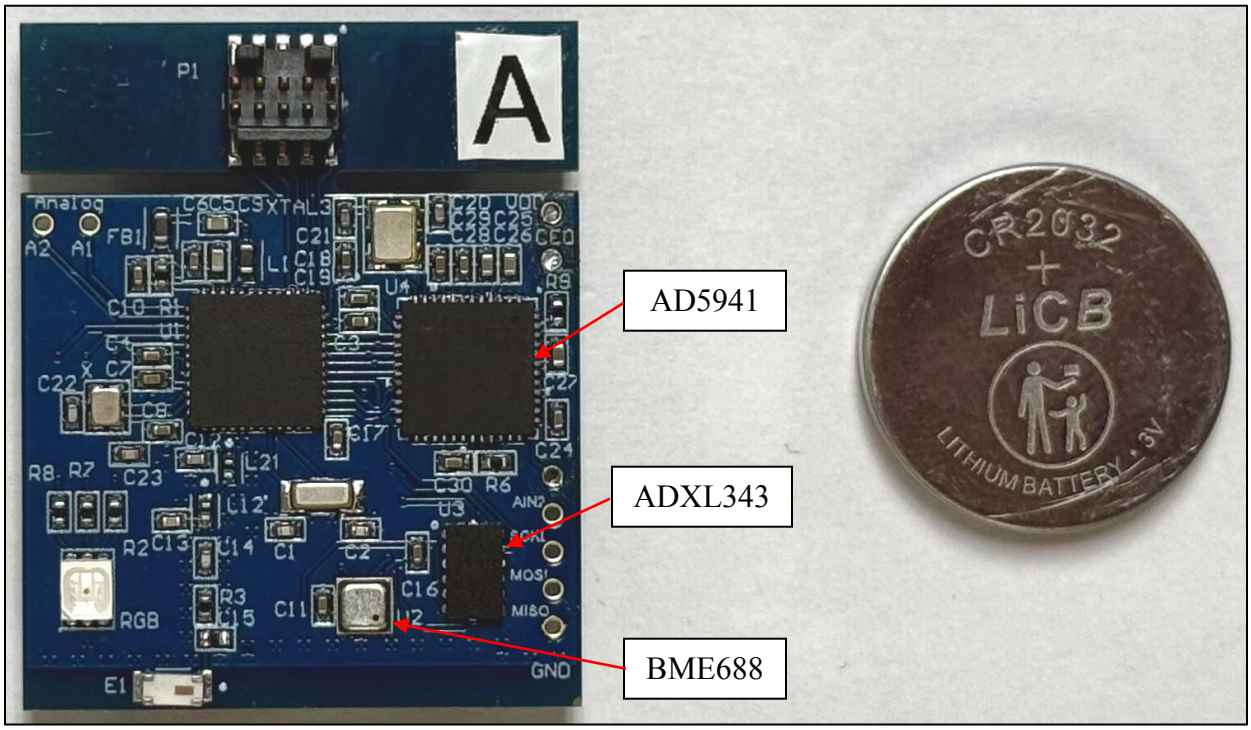

Figure 6: YSU Tag A next to a CR2032 battery.

The YSU Tag's firmware [31], written entirely in C, is uploaded to it via a Joint Test Action Group (JTAG) connection to a Texas Instruments (TI) MCU. To initiate the firmware upload, the MCU must be connected to a computer with TI's Integrated Development Environment (IDE), Code Composer Studio (CCS), via USB. Once CCS detects the MCU's connection to the computer, firmware uploads will be permitted.

The following subsections detail the sensors' configuration for this system and the BLE data packets. Lastly, the YSU Tag's complete BLE transmission structure will be examined.

3.2.1 BME688

Bosch, the manufacturer of the BME688, states that it is the first gas sensor integrated with Artificial Intelligence (AI). Its gas sensor can detect various gases such as Volatile Organic Compounds (VOCs), volatile sulfur compounds (VSCs), as well as gases like carbon monoxide and hydrogen at concentrations as low as parts per billion (ppb) [32]. Although the AI gas-sensing feature has not yet been implemented in this design, it could be utilized in future development to train different gases to be detected.

In CCS, the BME688's firmware outputs four different measurements: temperature in hundredths of degrees Celsius, barometric pressure in Pascals, relative humidity in thousandths of a percent, and gas resistance in ohms. Temperature is stored in 16 bits in memory, while the other three measurements are stored in 32 bits. Bosch provided an open-source firmware package for the BME688, reducing the overall development time. However, the four measurements will be transmitted via BLE, a function that Bosch did not provide. The data is split into one-byte chunks, bit-masked to ensure no overflow bits, and organized into a fourteen-element array of 8 bits as shown in Figure 7 to be sent to the BME688's measurement characteristic [19].

```
// Storing the BME688 data into an array to be sent to its measurement characteristic.
data[0] = bme.data.temperature & 0xff;
data[1] = (bme.data.temperature >> 8) & 0xff;
data[2] = bme.data.pressure & 0xff;
data[3] = (bme.data.pressure >> 8) & 0xff;
data[4] = (bme.data.pressure >> 16) & 0xff;
data[5] = (bme.data.pressure >> 24) & 0xff;
data[6] = bme.data.humidity & 0xff;
data[7] = (bme.data.humidity >> 8) & 0xff;
data[8] = (bme.data.humidity >> 16) & 0xff;
data[9] = (bme.data.humidity >> 24) & 0xff;
data[10] = bme.data.gas_resistance & 0xff;
data[11] = (bme.data.gas_resistance >> 8) & 0xff;
data[12] = (bme.data.gas_resistance >> 16) & 0xff;
data[13] = (bme.data.gas_resistance >> 24) & 0xff;
```

Figure 7: Storing raw BME688 measurement data.

3.2.2 ADXL343

The ADXL343 is a flexible and robust 3-axis accelerometer configurable for resolution, sensitivity, and offsets for measurement error [33]. Acceleration is measured with the ADXL343 using Micro-Electro-Mechanical Systems (MEMS) technology, meaning it combines micro-mechanical and electronic functions using silicon and semiconductor processes, which are compact, lightweight, energy-efficient, low-cost, reliable, and resistant to vibration and shock [34]. Higher sensitivity configurations are ideal for fall detections, where large and less precise acceleration values are required [35]. In contrast, lower sensitivity configurations would be suitable for vibrational detections, where smaller and more precise acceleration values are required [36].

In CCS, the ADXL343's firmware outputs tri-axial acceleration measurements in terms of gravitational acceleration (g), a constant defined as the gravitational acceleration near the surface of the Earth, equaling 9.81 m/s^2 [35]. With the ADXL343 oriented upwards (See Figure 6), the x and y axes reside on the horizontal plane, and the z-axis resides on the vertical plane. Each axis' acceleration measurement is stored in 16 bits, later split into a most-significant byte and a least-significant byte, with the increase in bit significance corresponding to higher measurement value. When read, acceleration data is stored in an eight-element array of 8 bits

as shown in Figure 8 to be sent to the ADXL343's measurement characteristic [19], [20], with two array elements initialized to zero that could output other data in future designs.

```
// Storing the ADXL343 data into an array to be sent to its measurement characteristic.
data[0] = x_lsb;
data[1] = x_msb;
data[2] = y_lsb;
data[3] = y_msb;
data[4] = z_lsb;
data[5] = z_msb;
data[6] = 0x00;
data[7] = 0x00;
```

Figure 8: Storing raw ADXL343 measurement data.

3.2.3 AD5941

The AD5941 is a high-precision, impedance, and electrochemical front-end. It has numerous applications, such as potentiostat, skin and body impedance, voltammetry, and glucose monitoring [37]. The AD5941 generates a fixed frequency in this design and measures an external load's 2-wire impedance at the designated frequency. Impedance is measured through the AD5941's Discrete Fourier Transform (DFT) engine, returning its real and imaginary parts and later converted to its polarized form of magnitude and phase. To take this measurement, users must connect wires to pins CE0 and AIN2 on the YSU Tag and connect the other ends of the wires to an impedance load.

In CCS, the AD5941's firmware outputs the generated frequency for impedance measuring and the connected external load's polarized impedance components of magnitude and phase. All three measured values: frequency, magnitude, and phase, are stored in 16 bits of memory, later split into a most significant byte and a least significant byte, with the increase in bit significance corresponding to higher measurement value. When the reading process is complete, frequency and impedance data are stored in a fourteen-element array of 8 bits as shown in Figure 9 to be sent to the AD5941's measurement characteristic [21], with eight array elements initialized to zero that could output other data in future designs.

```
// Storing the AD5941 data into an array to be sent to its measurement characteristic.
data[0] = freqArr[0]; // MSB
data[1] = freqArr[1]; // LSB
data[2] = 0x00;
data[3] = 0x00;
data[4] = mag[0]; // MSB
data[5] = mag[1]; // LSB
data[6] = 0x00;
data[7] = 0x00;
data[8] = 0x00;
data[9] = 0x00;
data[10] = 0x00;
data[11] = 0x00;
data[12] = phase[0]; // MSB
data[13] = phase[1]; // LSB
```

Figure 9: Storing raw AD5941 measurement data.

3.2.4 BLE Data Organization

The YSU Tag is a BLE peripheral, meaning it is connectable to a BLE central. Once the central has indicated that it wants to connect to the peripheral and the connection is established, the peripheral will start sending previously locked data that was unavailable before the connection was established. Additionally, the device roles of central and peripheral change to client and server, respectively, when they establish a connection. The locked data is referred to as attributes, which are data structures that store information about the server and are categorized as either services, characteristics, or descriptors. Characteristics are what hold attainable values, such as sensor measurements. Also, characteristics are assigned different permissions, such as reading and writing. The BLE protocol governs the organization of attributes in a table-like structure, where each attribute is assigned a handle. Handles, analogous to array indices in programming, start with the 16-bit hexadecimal value of 0x0001 for the first attribute and increment by one until all the server's attributes are accounted for. This structure is known as the Generic Attribute Profile (GATT), and each attribute will always be assigned the same handle as long as the server's BLE stack remains unchanged [6], [26]. This

consistency in handle assignments will affect how each sensor's characteristics are read from and written to. The YSU Tag's attribute table is shown in Table 1.

Attribute Description	Handle(s)
Generic Access Profile (GAP)	0x0001-0x000C
Primary Service Declaration	0x000D
Characteristic Declaration	0x000E
BME688 Measurement	0x000F
Client Characteristic Configuration	0x0010
Characteristic User Description	0x0011
Characteristic Declaration	0x0012
BME688 Configuration	0x0013
Characteristic User Description	0x0014
Characteristic Declaration	0x0015
BME688 Period	0x0016
Characteristic User Description	0x0017
Primary Service Declaration	0x0018
Characteristic Declaration	0x0019
ADXL343 Measurement	0x001A
Client Characteristic Configuration	0x001B
Characteristic User Description	0x001C
Characteristic Declaration	0x001D
ADXL343 Configuration	0x001E
Characteristic User Description	0x001F
Characteristic Declaration	0x0020
ADXL343 Period	0x0021
Characteristic User Description	0x0022
Primary Service Declaration	0x0023
Characteristic Declaration	0x0024
AD5941 Measurement	0x0025
Client Characteristic Configuration	0x0026
Characteristic User Description	0x0027
Characteristic Declaration	0x0028
AD5941 Configuration	0x0029
Characteristic User Description	0x002A
Characteristic Declaration	0x002B
AD5941 Period	0x002C
Characteristic User Description	0x002D

Table 1: YSU Tag – BLE attributes.

The attributes of the Generic Access Profile (GAP) were condensed in Table 1 since this is outside the focus of this thesis. The GAP's attributes store data such as the device name,

connection parameters, etc. The noteworthy attributes for this design include each sensor's measurement and configuration characteristics, with their corresponding handle values, which will be utilized to perform specific tasks in both embedded software applications.

3.3 IoT Gateway – xPico 250 Evaluation Kit

The xPico 250 Evaluation Kit, as shown in Figure 10, was chosen for this system because it meets all design requirements: supports at least four concurrent BLE connections, provides a compatible SDK for customized development, offers serial port communication for the terminal application, and features an HTTP server for the website.

Figure 10: xPico 250 Evaluation Kit.

Robustness and security are two key priorities of the SDK, which performs a validation check on the developer uploading firmware to the xPico by verifying their signature. If the firmware is not signed with the correct private key, it will fail to upload. The SDK includes an extensive library of sample programs that can be modified, thereby hastening the development stage. The terminal and website applications originate from sample programs and will be merged into a unified

program, “TagView” [38]. By design, TagView’s terminal and website applications will be functionally independent, allowing users to choose between them based on convenience.

The SDK’s library includes the sample terminal program, a serial application that connects to and interfaces with BLE peripherals. Once a connection is established with a peripheral, the program can perform operations such as reading and writing to the peripheral’s BLE characteristics. The sample program is limited to outputting only raw hexadecimal data from BLE characteristics, and the interface is inefficient, as users must enter many commands in the terminal to accomplish what could be reduced to just two commands.

The sample website is an open-source application available on Lantronix’s GitHub profile. It is designed to scan BLE heart rate monitors and display their real-time measurements on a customized website hosted on the xPico 250’s embedded HTTP server. Since the sample website was developed with an obsolete SDK version, attempting to compile the entire program with the latest SDK version resulted in unresolvable errors attributed to the outdated SDK version it relied on. Instead of modifying the base website package, it is used as a structural reference. Custom functions will be written to ensure sensor data arrives at the correct destination. The open-source Bootstrap framework will also enable dynamic functionality for the website’s user interface (UI) by adding stylized buttons to initiate specific functions.

TagView will address the shortcomings of the sample programs. TagView’s terminal will display cohesive sensor measurements for each connected YSU Tag based on the corresponding BLE characteristics. Also, the inefficiency of the sample program will be eliminated, reducing manual intervention to only a starting command and a stopping command. TagView’s website will be similar to the sample website's vision: collecting real-time sensor data from specific BLE peripherals. Instead of heart rate monitors, however, the website will be updated to support only

YSU Tags, acquiring cohesive sensor measurements at periodic intervals. The following sections will thoroughly examine each application, evaluating the development and derivation of key functions.

3.3.1 Terminal Development

TagView's terminal application is written entirely in C. Each step of its algorithm is defined in one function, "TagView_CLI", which will be called with a single command, "start", in the serial terminal emulator of choice. The following algorithm is required for the TagView terminal:

1. Connect to all powered-on and in-range YSU Tags.
2. Enable sensor measurement readouts.
3. Read and translate sensor measurements.
4. Print sensor measurements to the terminal.
5. Repeat step 4 and stop the program when the user enters any key on the keyboard.

Before writing the algorithm, however, there must be a global struct that stores essential data for each YSU Tag. Since there are four YSU Tags, each having unique properties at any given time, a four-element array of this global structure is initialized. The parameters of this global structure will be showcased later, with most being initialized to zero until data is collected. The declaration of this global struct, "YSU_Tag", is shown in Figure 11, and the initialization of a four-element array of "YSU_Tag" is shown in Figure 12.

```
// Global struct for storing important data about each YSU Tag.
struct YSU_Tag {
    uint8_t address[6]; // Device public MAC address (6 bytes).
    char label; // (A,B,C,D,...).
    uint16_t connection_id; // Connection ID tracker (initialized to 0).
    int tmpC_Int, tmpC_Frac, tmpF_Int, tmpF_Frac, prsInt, prsFrac, humInt, humFrac, gasInt, gasFrac; // BME688 data.
    int xInt, xFrac, yInt, yFrac, zInt, zFrac; // ADXL343 data.
    int frqInt, frqFrac, magInt, magFrac, phsInt, phsFrac; // AD5941 data.
    bool neg_tmpC, neg_tmpF, negX, negY, negZ, negPhs; // Booleans for printing negative signs.
};
```

Figure 11: Global declaration of "YSU_Tag" structure.

```
// Creating four instances of "YSU_Tag" since there are four devices.
struct YSU_Tag tags[] = {
    {
        // YSU Tag A: "84:C6:92:FD:EF:36"
        {0x84, 0xC6, 0x92, 0xFD, 0xEF, 0x36}, 'A', // MAC address and label.
    },
    {
        // YSU Tag B: "84:C6:92:FD:EF:44"
        {0x84, 0xC6, 0x92, 0xFD, 0xEF, 0x44}, 'B', // MAC address and label.
    },
    {
        // YSU Tag C: "B0:D2:78:65:74:2D"
        {0xB0, 0xD2, 0x78, 0x65, 0x74, 0x2D}, 'C', // MAC address and label.
    },
    {
        // YSU Tag D: "B0:D2:78:65:74:2F"
        {0xB0, 0xD2, 0x78, 0x65, 0x74, 0x2F}, 'D', // MAC address and label.
    }
};
```

Figure 12: Initialization for a four-element array of "YSU_Tag" structures.

The first step of TagView's terminal algorithm requires a modification of a function from the sample program. This function would connect to the corresponding BLE device's specified Media Access Control (MAC) address. For example, the user would enter the following command in the terminal to connect to a BLE device with the MAC address of "00:11:22:33:44:55": "connect 001122334455". Instead of performing this operation on one device at a time, the modified connection function will attempt to connect to all powered-on and in-range YSU Tags, referring to their hardware-defined MAC addresses. Table 2 shows the hardware-defined MAC addresses of each YSU Tag.

Device Name	Hardware-Defined MAC Address
YSU Tag A	84:C6:92:FD:EF:36
YSU Tag B	84:C6:92:FD:EF:44
YSU Tag C	B0:D2:78:65:74:2D
YSU Tag D	B0:D2:78:65:74:2F

Table 2: Hardware-defined MAC addresses of YSU Tags.

The modified connection function will first attempt to connect to YSU Tag A and display status messages in the terminal based on the connection result. An SDK-defined BLE connection callback function assigns connection identification numbers to connected BLE devices; therefore, if the connection to the requested YSU Tag is successful, it will store the resulting connection ID in its struct parameter, "connection_id", from Figure 11. If the connection to the requested YSU Tag is unsuccessful, the connection ID value of 0 will be stored in its corresponding struct parameter, which will be used for additional logic in later processes. This process is completed in alphabetical order of the tag's later label, starting with YSU Tag A and ending with YSU Tag D. Additionally, delay functions were utilized to prevent the program from moving on to the next device too quickly and prevent connection failures, as well as implementing connection status messages to display in the terminal. Step 1 of the TagView's terminal algorithm is shown in Figure 13.

```
// Loop over each tag and attempt connection.
for (size_t i = 0; i < num_tags; i++) {
    struct gatt_connect_context gcc = { 0 };
    char msgBuffer[128];

    stream_printf(os, "\r\n"); // For visual enhancement.

    // Log that connection attempt is starting.
    snprintf(msgBuffer, sizeof(msgBuffer), "Connecting to YSU Tag %c...", tags[i].label);
    ltrx_write_user_message(lwumi, LTRX_USER_MESSAGE_SEVERITY__INFORMATIONAL, msgBuffer);

    // Wait before attempting the connection and for timed messages.
    ltrx_thread_sleep(500);

    // Initiate connection to the specified YSU Tag.
    // The gatt connect function will return a nonzero result if the targeted device is already.
    // connected to the xPico, in that case move to the next iteration immediately.
    if (ltrx_gatt_connect(tags[i].address, BT_TRANSPORT__LE, BLE_ADDRESS_TYPE__PUBLIC, gatt_connect_callback, &gcc) != 0) {
        snprintf(msgBuffer, sizeof(msgBuffer), "Already connected to YSU Tag %c.", tags[i].label);
        ltrx_write_user_message(lwumi, LTRX_USER_MESSAGE_SEVERITY__ERROR, msgBuffer);
        continue;
    }

    // Wait some time if the connection has not been established yet.
    uint32_t waitTimeMs = 2000; // 2 seconds.
    uint32_t elapsedMs = 0; // Accumulator for time elapsed.
    uint32_t stepMs = 100; // 100 ms.
    while (!gcc.complete && (elapsedMs < waitTimeMs)) {
        ltrx_thread_sleep(stepMs);
        elapsedMs += stepMs;
    }

    // If still not complete, cancel the connection and immediately move to the next iteration.
    if (!gcc.complete && ltrx_gatt_connect_cancel(tags[i].address)) {
        snprintf(msgBuffer, sizeof(msgBuffer), "Connection failed for YSU Tag %c.", tags[i].label);
        ltrx_write_user_message(lwumi, LTRX_USER_MESSAGE_SEVERITY__ERROR, msgBuffer);
        tags[i].connection_id = 0;
        continue;
    }

    // If made it this far, the connection has succeeded.
    snprintf(msgBuffer, sizeof(msgBuffer), "Connected to YSU Tag %c.", tags[i].label);
    ltrx_write_user_message(lwumi, LTRX_USER_MESSAGE_SEVERITY__INFORMATIONAL, msgBuffer);
    tags[i].connection_id = gcc.connection_id;

    // Wait before starting the next iteration for timed messages.
    ltrx_thread_sleep(500);
}
```

Figure 13: Terminal connection process to four YSU Tags.

The second step of TagView's terminal algorithm requires enabling sensor measurement readouts. First, the connection status of each YSU Tag is determined by reading its connection ID struct parameter, with a non-zero ID indicating it is connected. Following connection verification, a one-byte value of "0x01" is written to each connected device's corresponding sensor configuration characteristics using the SDK-defined BLE write function. Since there are three sensors on the YSU Tag, each device has three characteristic writes. The feature of enabling each sensor was previously configured in the YSU Tag's firmware to conserve power by adding user control for enabling or disabling sensors based on their necessity at a given time. Additionally, users can disable sensor measurement readouts by writing a one-byte value of

“0x00” to the requested sensors’ BLE configuration characteristics. This feature of disabling sensors will not be implemented in this design, but it could be a future addition to increase device power efficiency.

Referring to Table 1, the handles associated with the sensor configuration characteristics are 0x0013, 0x001E, and 0x0029. This process is implemented in a separate function named “enableSensors”, the full definition of which is shown in Figure 14. Once the “enableSensors” function has been called in “TagView_CLI”, there will be an added delay of three seconds before moving on to the next step of the algorithm, which is shown in Figure 15.

```
// Enables sensor reads by writing to the YSU Tag's BLE configuration characteristics.
void enableSensors()
{
    uint8_t data = 0x01; // Writing 0x01 to the sensor configs enables sensor measurement reads.

    // Inner loop writes to each device's handles, outer loop switches to the next YSU Tag.
    for (size_t i = 0; i < num_tags; i++) {
        if (tags[i].connection_id == 0) { // If the specified tag is not connected, skip to the next device.
            continue;
        }
        else {
            for (size_t j = 0; j < num_cHandles; j++) {
                // SDK's BLE characteristic write function. Writing 0x01 to each sensor's configuration characteristic.
                ltrx_gatt_write(tags[i].connection_id, cHandles[j], &data, 0, 1, WRITE_TYPE__NORMAL);
            }
        }
    }
}
```

Figure 14: “enableSensors” function definition.

```
// Enable sensor configuration characteristics.
stream_printf(os, "\nEnabling sensors...\r\n\n");
enableSensors();

// Wait before reading sensor measurement characteristics.
ltrx_thread_sleep(3000);
```

Figure 15: Terminal enabling sensor measurements process.

The third step of TagView’s terminal algorithm requires reading and translating sensor measurements from all connected YSU Tags. Immediately after the process highlighted in Figures 14 and 15, the “TagView_CLI” function will enter an infinite while loop, performing the sensor measurement characteristic reads from each connected YSU Tag. Similar to the second step of the terminal’s algorithm, the process of reading the requested YSU Tag’s sensor

measurements will only do so if its stored connection ID is non-zero. Following connection verification, the program will call a separate function, “getSensorData,” to acquire the specified device’s sensor measurements.

Two arguments are passed to the “getSensorData” function: the index of the YSU Tag array of structures and the handle value attempting to be read from. The same two arguments will then be passed to the SDK-defined BLE read function. Referring to Table 1, “getSensorData” will perform characteristic reads from three handles: 0x000F, 0x001A, and 0x0025. The resulting sensor measurement data fills a buffer array with raw hexadecimal data. Based on the requested handle, the buffer array is decoded. The BME688, ADXL343, and AD5941 data processing is shown in Figures 16, 17, and 18, respectively.

```
if (handle == BME688_MEASUREMENT && length >= 14)
{
    // First, raw BME688 data is read in from the buffer array.
    int16_t rawTmp = (int16_t)((buffer[1] << 8) | buffer[0]); // Units of hundredths of degC.
    int32_t rawPrs = (int32_t)((buffer[5] << 24) | (buffer[4] << 16) | (buffer[3] << 8) | buffer[2]); // Units of Pa.
    int32_t rawHum = (int32_t)((buffer[9] << 24) | (buffer[8] << 16) | (buffer[7] << 8) | buffer[6]); // Units of thousandths of %.
    int32_t rawGas = (int32_t)((buffer[13] << 24) | (buffer[12] << 16) | (buffer[11] << 8) | buffer[10]); // Units of ohms.

    /* Temperature ------------------------------------------------------------------------*/
    // Convert to hundredths of degF.
    int rawTmpF = (rawTmp * 9) / 5 + (32 * 100);

    // Checking if temperatures are negative and if so, store that fact and flip to positive for printing.
    tags[i].neg_tmpC = (rawTmp < 0);
    tags[i].neg_tmpF = (rawTmpF < 0);

    if (tags[i].neg_tmpC) {
        rawTmp = -rawTmp;
    }
    if (tags[i].neg_tmpF) {
        rawTmpF = -rawTmpF;
    }

    // Converting hundredths of degC to degC and extracting the integer and decimal parts.
    tags[i].tmpC_Int  = rawTmp / 100; // integer part
    tags[i].tmpC_Frac = rawTmp % 100; // fractional part (two digits)

    // Converting hundredths of degF to degF and extracting the integer and decimal parts.
    tags[i].tmpF_Int  = rawTmpF / 100; // integer part
    tags[i].tmpF_Frac = rawTmpF % 100; // fractional part (two digits)

    /* Pressure ---------------------------------------------------------------------------*/
    // Converting Pa to hPa and extracting the integer and decimal parts.
    //int scaledPress = rawPressure;
    tags[i].prsInt  = rawPrs / 100; // integer part
    tags[i].prsFrac = rawPrs % 100; // fractional part (two digits)

    /* Humidity ---------------------------------------------------------------------------*/
    // Converting thousandths of % to % and extracting the integer and decimal parts.
    tags[i].humInt  = rawHum / 1000; // integer part
    tags[i].humFrac = (rawHum % 1000) / 10; // fractional part (two digits)

    /* Gas Resistance ---------------------------------------------------------------------*/
    // Converting ohms to kOhms and extracting the integer and decimal parts.
    tags[i].gasInt  = rawGas / 1000; // integer part
    tags[i].gasFrac = (rawGas % 1000) / 10; // fractional part (two digits)
}
```

Figure 16: BME688 measurement characteristic translation.

```
else if (handle == ADXL343_MEASUREMENT && length >= 8)
{
    // First, raw accleration data is read in from the buffer array.
    int16_t rawX = (int16_t)((buffer[1] << 8) | buffer[0]);
    int16_t rawY = (int16_t)((buffer[3] << 8) | buffer[2]);
    int16_t rawZ = (int16_t)((buffer[5] << 8) | buffer[4]);

    // Shifting the bits over 3 spaces since this is how the ADXL343 firmware processes it.
    rawX >>= 3;
    rawY >>= 3;
    rawZ >>= 3;

    // Scaling down the data by the ADXL343's defined scaling factor.
    int16_t scale = 256; // 256 LSB/g

    // Extracting the first two decimal places so multipky by 100 and then scale it down.
    int scaledX = (rawX * 100) / scale;
    int scaledY = (rawY * 100) / scale;
    int scaledZ = (rawZ * 100) / scale;

    // Checking if numbers are negative and if so, store that fact and flip to positive for printing.
    tags[i].negX = (scaledX < 0);
    tags[i].negY = (scaledY < 0);
    tags[i].negZ = (scaledZ < 0);

    if (tags[i].negX) {
        scaledX = -scaledX;
    }
    if (tags[i].negY) {
        scaledY = -scaledY;
    }
    if (tags[i].negZ) {
        scaledZ = -scaledZ;
    }

    // Extracting the integer values.
    tags[i].xInt = scaledX / 100;
    tags[i].yInt = scaledY / 100;
    tags[i].zInt = scaledZ / 100;

    // Extracting the first two decimal places.
    tags[i].xFrac = scaledX % 100;
    tags[i].yFrac = scaledY % 100;
    tags[i].zFrac = scaledZ % 100;
}
```

Figure 17: ADXL343 measurement characteristic translation.

```
else if (handle == AD5941_MEASUREMENT && length >= 14)
{
    // First, raw impedance data is read in from the buffer array.
    int16_t rawFrq = ((buffer[0] << 8) | buffer[1]);
    int16_t rawMag = ((buffer[4] << 8) | buffer[5]);
    int16_t rawPhs = ((buffer[12] << 8) | buffer[13]);

    /* Frequency ------------------------------------------------------------------------*/
    // Extracting the integer and decimal parts.
    tags[i].frqInt = rawFrq; // integer part
    tags[i].frqFrac = rawFrq % 1; // fractional part (two digits)

    /* Magnitude ------------------------------------------------------------------------*/
    // Extracting the integer and decimal parts.
    tags[i].magInt = rawMag / 100; // integer part
    tags[i].magFrac = rawMag % 100; // fractional part (two digits)

    /* Phase ----------------------------------------------------------------------------*/
    // Checking if negative and if so, store that fact and flip to positive for printing.
    tags[i].negPhs = (rawPhs < 0);
    if (tags[i].negPhs) {
        rawPhs = -rawPhs;
    }

    // Extracting the integer and decimal parts.
    tags[i].phsInt = rawPhs / 100; // integer part
    tags[i].phsFrac = rawPhs % 100; // fractional part (two digits)
}
```

Figure 18: AD5941 measurement characteristic translation.

The fourth step of TagView's terminal algorithm requires printing sensor measurements to the terminal. Once all a specific YSU Tag's sensors have been read, the translated sensor measurements will be displayed in the terminal emulator of choice for real-time monitoring. Finally, the fifth step of the algorithm requires the process explained in the fourth step to repeat until the user enters any key on the keyboard into the terminal. This step was added due to user convenience; instead of rebooting the xPico 250 and reloading TagView in the terminal emulator, the user can quickly stop the repeated cycle of reading YSU Tags' sensor measurements. The fourth and fifth steps of the terminal algorithm are shown in Figure 19.

```
while (true) {
    stream_printf(os, "Press any key to stop:\r\n");

    // Inner loop writes to each device's handles, outer loop switches to the next device.
    for (size_t i = 0; i < num_tags; i++) {
        if (tags[i].connection_id == 0) { // If the specified tag is not connected, skip to the next device.
            continue;
        }
        else {
            stream_printf(os, "\nYSU Tag %c:\r\n\n", tags[i].label);
            for (size_t j = 0; j < num_mHandles; j++) {
                getSensorData(i, mHandles[j]);
            }

            // Printing the formatted BME688 data to the terminal.
            stream_printf(os, "BME688:\r\n");
            stream_printf(os, "  Temperature    = %s%d.%02d degF (%s%d.%02d degC)\r\n", (tags[i].neg_tmpF ? "-" : ""),
                          tags[i].tmpF_Int, tags[i].tmpF_Frac, (tags[i].neg_tmpC ? "-" : ""), tags[i].tmpC_Int,
                          tags[i].tmpC_Frac);
            stream_printf(os, "  Pressure       = %d.%02d hPa\r\n", tags[i].prsInt, tags[i].prsFrac);
            stream_printf(os, "  Humidity       = %d.%02d%%\r\n", tags[i].humInt, tags[i].humFrac);
            stream_printf(os, "  Gas Resistance = %d.%02d kOhms\r\n", tags[i].gasInt, tags[i].gasFrac);

            // Printing the formatted ADXL343 data to the terminal.
            stream_printf(os, "ADXL343:\r\n");
            stream_printf(os, "  x = %s%d.%02d g\r\n", (tags[i].negX ? "-" : ""), tags[i].xInt, tags[i].xFrac);
            stream_printf(os, "  y = %s%d.%02d g\r\n", (tags[i].negY ? "-" : ""), tags[i].yInt, tags[i].yFrac);
            stream_printf(os, "  z = %s%d.%02d g\r\n", (tags[i].negZ ? "-" : ""), tags[i].zInt, tags[i].zFrac);

            // Printing the formatted AD5941 data to the terminal.
            stream_printf(os, "AD5941:\r\n");
            stream_printf(os, "  Frequency = %d.%02d Hz\r\n", tags[i].frqInt, tags[i].frqFrac);
            stream_printf(os, "  Magnitude = %d.%02d Ohms\r\n", tags[i].magInt, tags[i].magFrac);
            stream_printf(os, "  Phase     = %s%d.%02d deg\r\n\n", (tags[i].negPhs ? "-" : ""), tags[i].phsInt,
                          tags[i].phsFrac);
        }
    }

    // Checks to see if user pressed any key in the terminal.
    if (ltrx_input_stream_peek_with_block_time(is, 100) >= 0)
    {
        ltrx_input_stream_read(is);
        break;
    }
}
```

Figure 19: Terminal sensor measurement readout process.

3.3.2 Website Development

Unlike TagView's terminal application, its website is written in several languages. This includes C for back-end development of the function handling, as well as JavaScript (JS), Cascading Style Sheets (CSS), and HyperText Markup Language (HTML) for front-end development of the UI on the website. TagView's website uses some of the same back-end functions as the terminal, simplifying its development process significantly. The following algorithm is required for the TagView website:

1. Connect to all powered-on and in-range YSU Tags.
2. Enable sensor measurement readouts.
3. Read and translate sensor measurements.

4. Display sensor measurements on the website.

Steps one through three of TagView's website algorithm are identical to the terminal's, implementing the same functions the terminal application uses. The fourth step of the website's algorithm requires displaying sensor measurements. A separate C file defines an Application Programming Interface (API) to transmit requested data to the xPico 250's embedded HTTP server. API requests are managed through Uniform Resource Identifiers (URIs). Figure 20 shows the "/getDAQ" URI endpoint on the first line of code, with steps one through three executing under this endpoint. The full URI is "/tagView/getDAQ" and will be requested when a button is asserted on the website.

```
if (memcmp(path, "/getDAQ", 7) == 0) { // Full URI path is "/tagView/getDAQ".
    bool firstElement = true; // For formatting the JSON array.

    webSetup(); // Calling from ble_gatt.c

    ltrx_ip_socket_send(socket, http_header, strlen(http_header), false);
    ltrx_ip_socket_send(socket, "[", 1, false); // Starting the JSON array.

    for (size_t i = 0; i < num_tags; i++) {
        if (tags[i].connection_id == 0) { // If the specified device is not connected, skip to the next one.
            continue;
        }

        if (firstElement == true) { // Format the JSON array with a leading comma if not the first element, else no comma.
            ltrx_ip_socket_send(socket, "{", 1, false);
            firstElement = false;
        }
        else {
            ltrx_ip_socket_send(socket, ",{", 2, false);
        }

        for (size_t j = 0; j < num_mHandles; j++) {
            getSensorData(i, mHandles[j]); // Calling from ble_gatt.c
        }
```

Figure 20: YSU Tag connection and sensor reading in the API.

Immediately following the process highlighted in Figure 20, each connected YSU Tag's structure data is stored in a JavaScript Object Notation (JSON) object. This JSON object can be transmitted over HTTP, making it a popular choice for establishing communication between the back-end and front-end sides of applications. This process is shown in Figures 21 and 22. This concludes the back-end development of the website.

```
/* Sending sensor data ------------------------------------------------------------*/
size = snprintf(buf, sizeof(buf), ",\"address\":\"%02X:%02X:%02X:%02X:%02X:%02X\"", // MAC Address
                tags[i].address[0], tags[i].address[1], tags[i].address[2],
                tags[i].address[3], tags[i].address[4], tags[i].address[5]);
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), "\"label\":\"%c\"", tags[i].label); // Label
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"tmpC_Int\":%d", tags[i].tmpC_Int); // Temperature in degC (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"tmpC_Frac\":%d", tags[i].tmpC_Frac); // Temperature in degC (decimal).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"tmpF_Int\":%d", tags[i].tmpF_Int); // Temperature in degF (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"tmpF_Frac\":%d", tags[i].tmpF_Frac); // Temperature in degF (decimal).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"prsInt\":%d", tags[i].prsInt); // Pressure in hPa (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"prsFrac\":%d", tags[i].prsFrac); // Pressure in hPa (decimal).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"humInt\":%d", tags[i].humInt); // Humidity in % (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"humFrac\":%d", tags[i].humFrac); // Humidity in % (decimal).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"gasInt\":%d", tags[i].gasInt); // Gas resistance in kOhms (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"gasFrac\":%d", tags[i].gasFrac); // Gas resistance in kOhms (decimal).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"xInt\":%d", tags[i].xInt); // x-axis acceleration in g (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"xFrac\":%d", tags[i].xFrac); // x-axis acceleration in g (decimal).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

size = snprintf(buf, sizeof(buf), ",\"yInt\":%d", tags[i].yInt); // y-axis acceleration in g (integer).
ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.
```

Figure 21: Sending sensor data into a JSON object, part 1.

```
        size = snprintf(buf, sizeof(buf), ",\"yFrac\":%d", tags[i].yFrac); // y-axis acceleration in g (decimal).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"zInt\":%d", tags[i].zInt); // z-axis acceleration in g (integer).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"zFrac\":%d", tags[i].zFrac); // z-axis acceleration in g (decimal).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"frqInt\":%d", tags[i].frqInt); // Frequency in Hz (integer).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"frqFrac\":%d", tags[i].frqFrac); // Frequency in Hz (decimal).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"magInt\":%d", tags[i].magInt); // Magnitude in ohms (integer).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"magFrac\":%d", tags[i].magFrac); // Magnitude in ohms (decimal).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"phsInt\":%d", tags[i].phsInt); // Phase in degrees (integer).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"phsFrac\":%d", tags[i].phsFrac); // Phase in degrees (decimal).
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"neg_tmpC\":%d", tags[i].neg_tmpC); // Boolean for printing negative sign for degC.
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"neg_tmpF\":%d", tags[i].neg_tmpF); // Boolean for printing negative sign for degF.
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"negX\":%d", tags[i].negX); // Boolean for printing negative sign for x-axis acc.
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"negY\":%d", tags[i].negY); // Boolean for printing negative sign for y-axis acc.
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"negZ\":%d", tags[i].negZ); // Boolean for printing negative sign for z-axis acc.
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        size = snprintf(buf, sizeof(buf), ",\"negPhs\":%d", tags[i].negPhs); // Boolean for printing negative sign for phase.
        ltrx_ip_socket_send(socket, buf, size, false); // Sending to JSON object.

        ltrx_ip_socket_send(socket, "}", 1, false); // Ending the JSON object.
    }
    ltrx_ip_socket_send(socket, "]", 1, true); // Ending the JSON array.
}
```

Figure 22: Sending sensor data into a JSON object, part 2.

The website's structure is written in HTML, relying on the open-source Bootstrap framework to enable dynamic functionality for the website's UI by adding stylized buttons to initiate specific functions and enhanced visual displays. Each YSU Tag's sensor data will be split into different columns, supporting up to four concurrent BLE connections. As previously mentioned, the requested path to the URI endpoint in the API is called when a user clicks a specific button on the website. This button is labeled "Scan YSU Tags" on the website and calls the "getSensors" function in the "main.js" file when it is clicked. The developed "tagView.html" file is shown in Figure 23.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>TagView</title>
  <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
  <link rel="icon" href="data:,">
</head>
<body>
  <div class="container my-3">
    <div id="sensorDisplay" class="row gx-3 gy-3"></div>
    <div class="row mt-3">
      <div class="col-12 text-center">
        <input
          type="button"
          class="btn btn-success btn-lg"
          value="Scan YSU Tags"
          onclick="getSensors()"
          style="border: 1px solid black;"
        >
      </div>
    </div>
  </div>

  <div id="templateDisplay" class="deviceBlock border rounded clearfix" style="display:none;">
    <div class="tagLabel float-right" style="font-size: 36px;margin-right:20px; font-weight: bold;"></div>
    <div><b>BME688:</b></div>
    <div class="temperature"></div>
    <div class="pressure"></div>
    <div class="humidity"></div>
    <div class="gasResistance"></div>
    <div><b>ADXL343:</b></div>
    <div class="xAcc"></div>
    <div class="yAcc"></div>
    <div class="zAcc"></div>
    <div><b>AD5941:</b></div>
    <div class="frequency"></div>
    <div class="impMag"></div>
    <div class="impPhs"></div>
  </div>

  <!-- JavaScript files should be placed before closing body tag for better performance -->
  <script src="js/bootstrap.bundle.min.js"></script>
  <script src="js/main.js"></script>
</body>
</html>
```

Figure 23: “tagView.html”.

Lastly, to bridge the connection between the API and the website, a JS file, “main.js”, will handle this task. There are two notable functions in “main.js”. First, the “formatValue” function will handle several sensor data conditions: if data was received at all, printing the sign of the sensor measurement if applicable, and formatting the hundredths place precision for each measurement. Second, the “displaySensors” function will get the JSON response, send data to the “formatValue” function, and format data strings to link to “tagView.html”. This process is shown in Figure 24.

```
// Formatting data based on if data was received from the API, if it is negative, and for hundreths place precision.
function formatValue(intVal, fracVal, isNegative = false) {
  if (intVal == null || fracVal == null) return "?";
  return (isNegative ? "-" : "") + intVal + "." + String(fracVal).padStart(2, "0");
}

function displaySensors(response) {
    var sensorDisplay = document.getElementById("sensorDisplay");
    sensorDisplay.innerHTML = ''; // Clear previous results

    response.forEach(value => {
        const template = document.getElementById("templateDisplay");
        const d = template.cloneNode(true);

        d.style.display = 'block';  // make visible
        d.classList.add('col-md-3'); // 4 items per row
        d.id = value.address;

        // Update sensor details:
        d.querySelector(".tagLabel").textContent = "YSU Tag " + value.label;

        d.querySelector(".temperature").textContent = "Temperature = "
                                                    + formatValue(value.tmpF_Int, value.tmpF_Frac, value.neg_tmpF) + " °F ("
                                                    + formatValue(value.tmpC_Int, value.tmpC_Frac, value.neg_tmpC) + " °C)";

        d.querySelector(".pressure").textContent = "Pressure = " + formatValue(value.prsInt, value.prsFrac) + " hPa";

        d.querySelector(".humidity").textContent = "Humidity = " + formatValue(value.humInt, value.humFrac) + " %";

        d.querySelector(".gasResistance").textContent = "Gas Resistance = " + formatValue(value.gasInt, value.gasFrac) + " kΩ";

        d.querySelector(".xAcc").textContent = "x = " + formatValue(value.xInt, value.xFrac, value.negX) + " g";

        d.querySelector(".yAcc").textContent = "y = " + formatValue(value.yInt, value.yFrac, value.negY) + " g";

        d.querySelector(".zAcc").textContent = "z = " + formatValue(value.zInt, value.zFrac, value.negZ) + " g";

        d.querySelector(".frequency").textContent = "Frequency = " + formatValue(value.frqInt, value.frqFrac) + " Hz";

        d.querySelector(".impMag").textContent = "Magnitude = " + formatValue(value.magInt, value.magFrac) + " Ω";

        d.querySelector(".impPhs").textContent = "Phase = " + formatValue(value.phsInt, value.phsFrac, value.negPhs) + "°";

        sensorDisplay.appendChild(d);
    });
}
```

Figure 24: Sensor data formatting functions in “main.js”.

CHAPTER 4 – RESULTS AND DISCUSSION

The system in Figure 25 was assembled for sensor DAQ in the TagView terminal and website applications. YSU Tags A and B are powered by a CR2032 coin cell battery, whereas two TI MCUs powered via USB provide power to YSU Tags C and D via JTAG. The xPico 250 Evaluation Kit is powered on via a power adapter, and the RS-232 end of an RS-232 to USB cable is connected to it for the terminal application, with the USB end connected to the computer. Lastly, two antennas are connected to it, one for Bluetooth and the other for Wi-Fi to be utilized with the website.

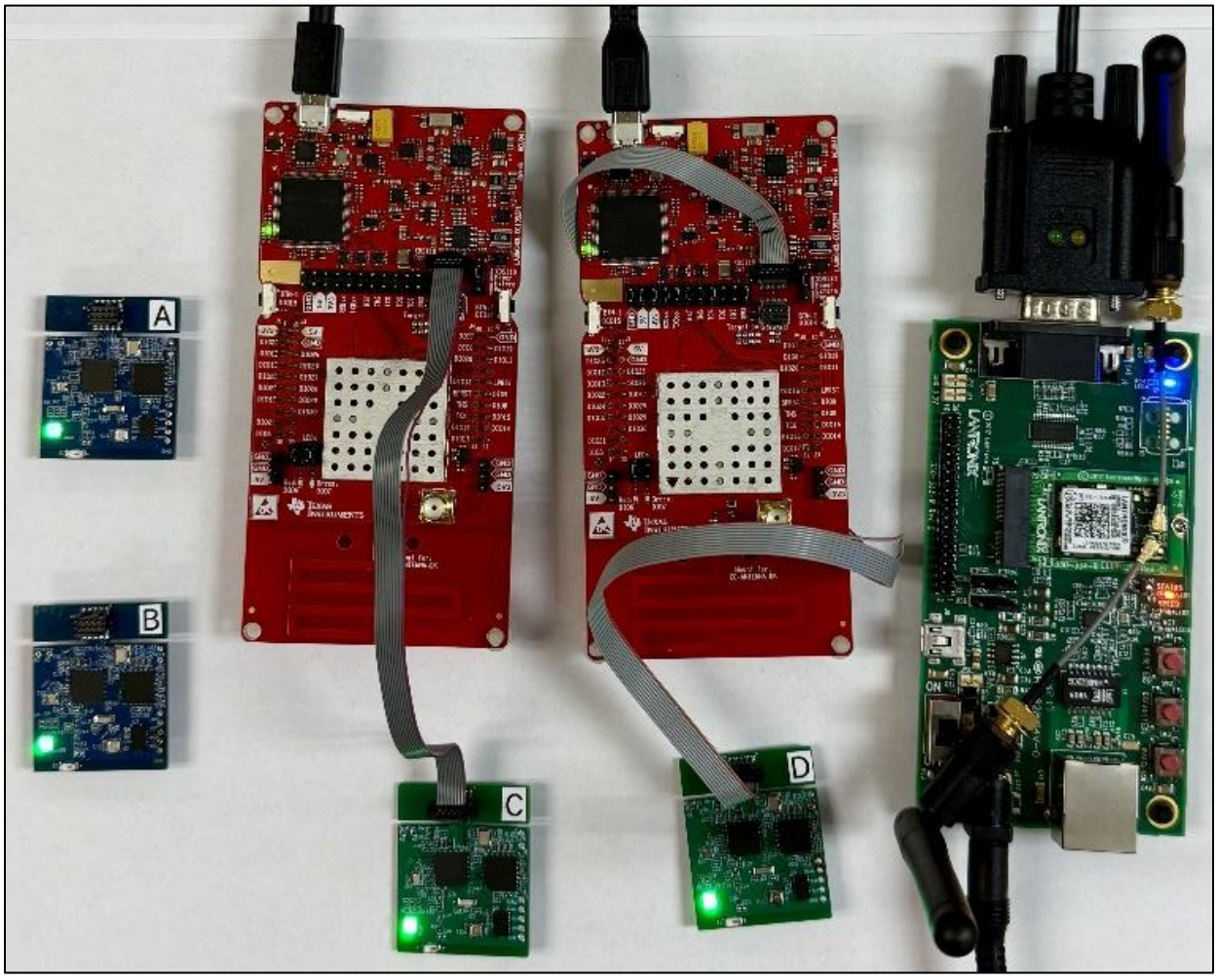

Figure 25: IoT network implementation.

The following subsections display and explain the output of both TagView applications: the serial terminal program and the dynamic website. Afterwards, the applications are compared, highlighting their successes and shortcomings.

4.1 Terminal Output

The TagView terminal application was developed to acquire real-time data concurrently from up to four YSU Tags and display it in a serial terminal emulator. Tera Term is the serial terminal emulator of choice, an open-source application widely used for serial port communication. Users write one command in the terminal to initiate the program: "start". Several status messages appear based on the connection result to each YSU Tag. Following the connection process, another status message indicates that sensors are being enabled. This start-up process is shown in Figure 26.

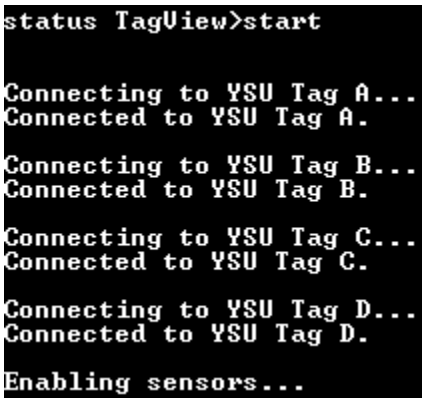

Figure 26: TagView – Terminal. Start-up sequence.

Once the start-up process is complete, the program will begin reading, translating, and printing sensor measurements. The program skips over YSU Tags that are not connected and will repeat the measurement read cycle once all connected devices have been read. Users can terminate the DAQ by pressing any key on the keyboard, which is shown in a message prompt in Figure 27. The DAQ from four YSU Tags is displayed in Figures 27 and 28.

```
Press any key to stop:

YSU Tag A:

BME688:
  Temperature     = 73.00 degF (22.78 degC)
  Pressure        = 978.30 hPa
  Humidity        = 37.74%
  Gas Resistance  = 104.80 kOhms
ADXL343:
  x = -0.01 g
  y = -0.05 g
  z = 0.98 g
AD5941:
  Frequency = 2000.00 Hz
  Magnitude = 0.00 Ohms
  Phase     = 0.00 deg


YSU Tag B:

BME688:
  Temperature     = 72.42 degF (22.46 degC)
  Pressure        = 978.22 hPa
  Humidity        = 37.92%
  Gas Resistance  = 44.80 kOhms
ADXL343:
  x = -0.01 g
  y = -0.02 g
  z = 1.01 g
AD5941:
  Frequency = 2000.00 Hz
  Magnitude = 0.00 Ohms
  Phase     = 0.00 deg
```

Figure 27: TagView – Terminal. DAQ from YSU Tags A and B.

```
YSU Tag C:

BME688:
  Temperature     = 75.07 degF (23.93 degC)
  Pressure        = 977.63 hPa
  Humidity        = 37.38%
  Gas Resistance  = 64.50 kOhms
ADXL343:
  x = 0.02 g
  y = -0.08 g
  z = 0.93 g
AD5941:
  Frequency = 2000.00 Hz
  Magnitude = 0.00 Ohms
  Phase     = 0.00 deg


YSU Tag D:

BME688:
  Temperature     = 75.97 degF (24.43 degC)
  Pressure        = 977.65 hPa
  Humidity        = 36.04%
  Gas Resistance  = 78.60 kOhms
ADXL343:
  x = -0.01 g
  y = 0.00 g
  z = 0.97 g
AD5941:
  Frequency = 2000.00 Hz
  Magnitude = 0.00 Ohms
  Phase     = 0.00 deg
```

Figure 28: TagView – Terminal. DAQ from YSU Tags C and D.

4.2 Website Output

The TagView website application was developed to acquire real-time data concurrently from up to four YSU Tags and display it on a customized website hosted on the xPico 250's HTTP server. This website can be accessed when a connection has been established between the xPico 250's Wi-Fi access point and virtually any Internet-accessible device, such as a computer or smartphone. When the user clicks "Scan YSU Tags", a similar process will happen as in the terminal application: attempting to connect to all powered-on and in-range YSU Tags, enabling sensor measurements, and reading and translating sensor measurements. The display shown in Figure 29 is a DAQ from four YSU Tags. The corresponding JSON response from the API is displayed in Figure 30, proving that sensor data was received from all four YSU Tags.

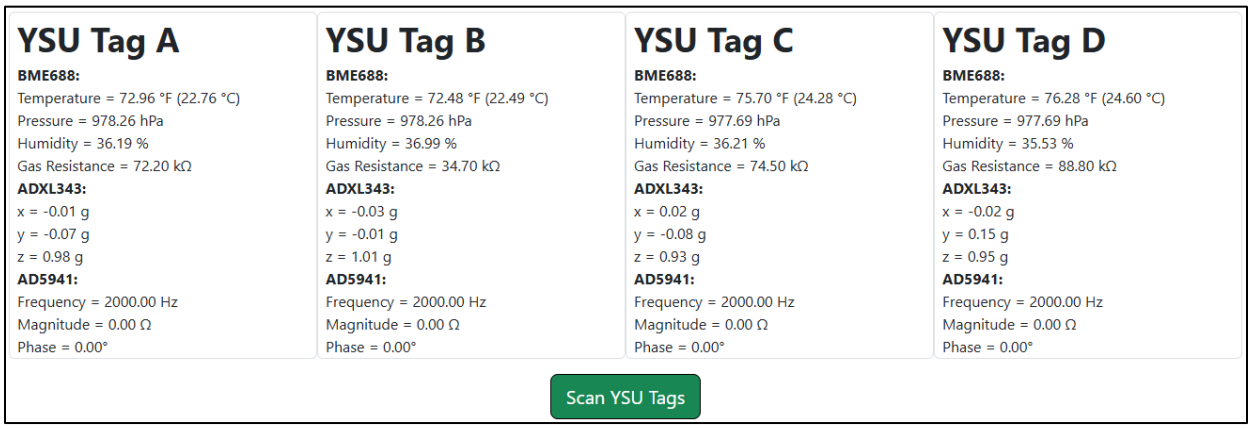

Figure 29: TagView – Website. DAQ from four YSU Tags.

JSON response: main.js:11
[{"address":"84:C6:92:FD:EF:36","label":"A","tmpC_Int":22,"tmpC_Frac":76,
"tmpF_Int":72,"tmpF_Frac":96,"prsInt":978,"prsFrac":26,"humInt":36,"humFr
ac":19,"gasInt":72,"gasFrac":20,"xInt":0,"xFrac":1,"yInt":0,"yFrac":7,"zI
nt":0,"zFrac":98,"frqInt":2000,"frqFrac":0,"magInt":0,"magFrac":0,"phsInt
":0,"phsFrac":0,"neg_tmpC":0,"neg_tmpF":0,"negX":1,"negY":1,"negZ":0,"neg
Phs":0},
{"address":"84:C6:92:FD:EF:44","label":"B","tmpC_Int":22,"tmpC_Frac":49,"
tmpF_Int":72,"tmpF_Frac":48,"prsInt":978,"prsFrac":26,"humInt":36,"humFra
c":99,"gasInt":34,"gasFrac":70,"xInt":0,"xFrac":3,"yInt":0,"yFrac":1,"zIn
t":1,"zFrac":1,"frqInt":2000,"frqFrac":0,"magInt":0,"magFrac":0,"phsInt":
0,"phsFrac":0,"neg_tmpC":0,"neg_tmpF":0,"negX":1,"negY":1,"negZ":0,"negPh
s":0},
{"address":"B0:D2:78:65:74:2D","label":"C","tmpC_Int":24,"tmpC_Frac":28,"
tmpF_Int":75,"tmpF_Frac":70,"prsInt":977,"prsFrac":69,"humInt":36,"humFra
c":21,"gasInt":74,"gasFrac":50,"xInt":0,"xFrac":2,"yInt":0,"yFrac":8,"zIn
t":0,"zFrac":93,"frqInt":2000,"frqFrac":0,"magInt":0,"magFrac":0,"phsInt"
:0,"phsFrac":0,"neg_tmpC":0,"neg_tmpF":0,"negX":0,"negY":1,"negZ":0,"negP
hs":0},
{"address":"B0:D2:78:65:74:2F","label":"D","tmpC_Int":24,"tmpC_Frac":60,"
tmpF_Int":76,"tmpF_Frac":28,"prsInt":977,"prsFrac":69,"humInt":35,"humFra
c":53,"gasInt":88,"gasFrac":80,"xInt":0,"xFrac":2,"yInt":0,"yFrac":15,"zI
nt":0,"zFrac":95,"frqInt":2000,"frqFrac":0,"magInt":0,"magFrac":0,"phsInt
":0,"phsFrac":0,"neg_tmpC":0,"neg_tmpF":0,"negX":1,"negY":0,"negZ":0,"neg
Phs":0}]

Figure 30: JSON response – DAQ from four YSU Tags.

When fewer than four YSU Tags are connected, the website should display blank values for each sensor's measurement to indicate to users that the specified device is not connected. In this case, a question mark will appear. Figure 31 shows that when the xPico 250 is connected to YSU Tags A and B, it should perform DAQs normally from those devices. However, since YSU Tags C and D are not connected, it displays question marks next to their sensor measurements.

Figure 31: TagView – Website. DAQ from two YSU Tags.

4.3 Discussion of Results

Overall, both applications resulted in successful DAQs from YSU Tags. The terminal application implements two commands: one for starting the DAQ and the other for stopping it. On the other hand, the website only implements one command, which is to begin the DAQ. Currently, the terminal application's DAQ is faster than the website's, likely due to the API's algorithm, which requires all connected YSU Tags' sensor measurements to be read before displaying all the data on the website. Additionally, the terminal outputs status messages at various stages of the program, whereas the website does not. However, the website has the potential to be a more user-friendly experience with continued development, as the UI already has enhanced visuals, whereas the terminal does not. For instance, the website could be accessed over the Internet; currently, it can only be accessed through a Wi-Fi connection to the xPico 250's access point.

Overall, both applications could be significantly improved in speed and user experience, since they are new additions to the design. This features a significant improvement from the previous work, which could interface with only one YSU Tag at a time.

CHAPTER 5 – CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis successfully demonstrated the design, development, and implementation of two software applications, a serial terminal program and a dynamic website, for real-time DAQ from up to four custom BLE peripherals using an IoT gateway. Utilizing the xPico 250 Evaluation Kit and its compatible SDK, the proposed system achieved concurrent BLE connectivity, overcoming the previous design's limitation of single-device communication.

The terminal application provides convenient local access through serial communication to a computer, while the dynamic website offers cross-platform accessibility and enhanced visual representation. Both applications effectively interface and acquire data from all three sensors on the YSU Tag: BME688, ADXL343, and AD5941. These sensor measurements include temperature, relative humidity, barometric pressure, gas resistance, tri-axial acceleration, and impedance.

Compared to previous work, this thesis enhances the system by enabling multi-device communication, centralized control, and dual-accessibility interfaces. The results validate the feasibility of using an embedded IoT gateway as a robust, secure, and flexible solution for managing custom sensor networks.

5.2 Future Work

Additional modifications could be implemented on the TagView website. For instance, adding support to facilitate data exportation from the website for a data analysis project, comparing each YSU Tag's sensor measurements to traditional measurement methods, and composing necessary firmware calibrations. Also, a system could be devised to have all connected YSU Tags communicate with each other through the xPico 250, something that was not possible before

integrating the gateway into the IoT network. This could benefit future development, as certain measurement thresholds could trigger other devices to perform specific functions. Additionally, more URI endpoints could be created in the website's API to handle different conditions and inputs, such as a stop button to terminate the DAQ or to disable specific sensors. Furthermore, it may be worthwhile to transfer functionality from the previously designed smartphone app over to the website, its standout feature being that it has graphs to log each sensor's measurements. Lastly, support for website remote access could be introduced by registering a domain name, since it can only be accessed by connecting to the xPico 250's Wi-Fi access point.

Regarding the YSU Tag, the AD5941's impedance measurements output zero for both magnitude and phase. However, this should require minimal firmware adjustments as the correct user-configured generated frequency is displayed. Furthermore, the AI gas-sensing feature of the BME688 could be implemented to detect specific gases. Also, the sensors' period characteristics could be investigated, as this supposedly controls the speed of the corresponding sensor's measurement readouts. Lastly, modifying the YSU Tag's BLE stack could optimize its power efficiency by altering its advertisement and removing unnecessary attributes.

REFERENCES

[1] S. N. Swamy and S. R. Kota, "An Empirical Study on System Level Aspects of Internet of Things (IoT)," *IEEE Access*, vol. 8, pp. 188082–188134, 2020, doi: 10.1109/ACCESS.2020.3029847.

[2] "Current IoT Forecast Highlights," TransformaInsights.com. Accessed: Aug. 27, 2024. [Online]. Available: https://transformainsights.com/research/forecast/highlights

[3] S. Lamsal, A. Uya, S. Itapu, F. X. Li, P. Cortes, and V. Borra, "Frequency selective asymmetric coupled-fed (ACS) antenna using additive manufacturing," *Memories - Materials, Devices, Circuits and Systems*, vol. 8, p. 100111, Aug. 2024, doi: 10.1016/J.MEMORI.2024.100111.

[4] A. Uya, S. Lamsal, S. Itapu, … F. L.-2023 I. 23rd, and undefined 2023, "Design and Fabrication of Terahertz (THz) Antenna Using Aerosol Jet Printing," *ieeexplore.ieee.org*, Accessed: Apr. 12, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10231203/

[5] A. Islam, D. Adu-Gyamfi, S. Itapu, F. X. Li, P. Cortes, and V. Borra, "Design and Fabrication of a Terahertz Antenna Using Two-Photon Polymerization," in *2024 IEEE Nanotechnology Materials and Devices Conference (NMDC)*, IEEE, 2024, pp. 169–174.

[6] C. Gomez, J. Oller, and J. Paradells, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," *Sensors*, vol. 12, no. 9, pp. 11734–11753, Aug. 2012, doi: 10.3390/s120911734.

[7] J. Villamizar, V. Borra, and F. Li, "Building a fully digital controller with Altera Cyclone FPGA and Nios Processor for Induction Heating application," *Discover Electronics*, vol. 1, no. 1, p. 27, 2024.

[8] Y. Sapkota, S. Devkota, V. Borra, P. Cortes, S. Itapu, and F. Li, "Harmonic content analysis of a soft starting variable frequency motor drive based on FPGA," in *2023 IEEE 3rd International Conference on Sustainable Energy and Future Electric Transportation (SEFET)*, IEEE, 2023, pp. 1–5.

[9] "NIHF Inductee Jaap C. Haartsen Invented Bluetooth Wireless Technology," Invent.org. Accessed: Aug. 27, 2024. [Online]. Available: https://www.invent.org/inductees/jaap-c-haartsen

[10] "About Us," Bluetooth® Technology Website. Accessed: Aug. 25, 2024. [Online]. Available: https://www.bluetooth.com/about-us/

[11] "About Us – Vision and Mission," Bluetooth® Technology Website. Accessed: Aug. 25, 2024. [Online]. Available: https://www.bluetooth.com/about-us/vision/

[12] "Nokia builds a better Bluetooth - Technology - International Herald Tribune," The New York Times. Accessed: Aug. 27, 2024. [Online]. Available: https://www.nytimes.com/2006/10/03/technology/03iht-nokia.3018027.html

[13] "Covered Core Package version: 4.0 Current Master TOC Specification Volume 0," 2010. [Online]. Available: http://www.bluetooth.com

[14] "Learn About Bluetooth – Bluetooth Technology Overview," Bluetooth® Technology Website. Accessed: Aug. 25, 2024. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/

[15] A. Barua, M. A. Al Alamin, Md. S. Hossain, and E. Hossain, "Security and Privacy Threats for Bluetooth Low Energy in IoT and Wearable Devices: A Comprehensive

Survey," *IEEE Open Journal of the Communications Society*, vol. 3, pp. 251–281, 2022, doi: 10.1109/OJCOMS.2022.3149732.
[16] "2024 Bluetooth Market Update," Bluetooth® Technology Website. Accessed: Aug. 27, 2024. [Online]. Available: https://www.bluetooth.com/2024-market-update/
[17] K. Nyako, S. Devkota, F. Li, and V. Borra, "Building Trust in Microelectronics: A Comprehensive Review of Current Techniques and Adoption Challenges," Nov. 01, 2023, *Multidisciplinary Digital Publishing Institute (MDPI)*. doi: 10.3390/electronics12224618.
[18] K. Nyako, U. Dhakal, F. Li, and V. Borra, "Trusted microelectronics: reverse engineering chip die using U-Net convolutional network," *Engineering Research Express*, vol. 6, no. 4, Dec. 2024, doi: 10.1088/2631-8695/ad7c06.
[19] R. J. Yarwood, "Multi-Sensor BLE Platform Using TI Wireless MCU and Mobile Application," 2022. Accessed: Jan. 03, 2025. [Online]. Available: http://rave.ohiolink.edu/etdc/view?acc_num=ysu1651521574915615
[20] J. Garretto, "Bluetooth Low Energy Communication for Multi-Sensor Applications Design and Analysis," 2022. Accessed: Jan. 03, 2025. [Online]. Available: http://rave.ohiolink.edu/etdc/view?acc_num=ysu167091921724991
[21] B. M. Kuzior, "Firmware Development and Applications of a Multi-Sensor Bluetooth Low Energy Peripheral," 2023. Accessed: Jan. 03, 2025. [Online]. Available: http://rave.ohiolink.edu/etdc/view?acc_num=ysu1702669149346863
[22] B. Kuzior, V. Borra, F. Li, B.-W. Park, and P. Cortes, "Impedance Measurements Using a High Precision, Low Power Analog Front End," in *243rd ECS Meeting with the 18th International Symposium on Solid Oxide Fuel Cells (SOFC-XVIII)*, ECS, 2023.
[23] A. Glória, F. Cercas, and N. Souto, "Design and implementation of an IoT gateway to create smart environments," *Procedia Comput Sci*, vol. 109, pp. 568–575, 2017, doi: 10.1016/j.procs.2017.05.343.
[24] C. Liu, Z. Su, X. Xu, and Y. Lu, "Service-oriented industrial internet of things gateway for cloud manufacturing," *Robot Comput Integr Manuf*, vol. 73, Feb. 2022, doi: 10.1016/j.rcim.2021.102217.
[25] T. Lojka, M. Bundzel, and I. Zolotová, "Industrial Gateway for Data Acquisition and Remote Control," *Acta Electrotechnica et Informatica*, vol. 15, no. 2, pp. 43–48, Jun. 2015, doi: 10.15546/aeei-2015-0017.
[26] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino, and D. Formica, "Performance evaluation of bluetooth low energy: A systematic review," Dec. 13, 2017, *MDPI AG*. doi: 10.3390/s17122898.
[27] R. De Fazio, M. De Vittorio, and P. Visconti, "A BLE-Connected Piezoresistive and Inertial Chest Band for Remote Monitoring of the Respiratory Activity by an Android Application: Hardware Design and Software Optimization," *Future Internet*, vol. 14, no. 6, Jun. 2022, doi: 10.3390/fi14060183.
[28] G. Alfian, M. Syafrudin, M. F. Ijaz, M. A. Syaekhoni, N. L. Fitriyani, and J. Rhee, "A personalized healthcare monitoring system for diabetic patients by utilizing BLE-based sensors and real-time data processing," *Sensors (Switzerland)*, vol. 18, no. 7, Jul. 2018, doi: 10.3390/s18072183.
[29] J. Hughes, J. Yan, and K. Soga, "DEVELOPMENT OF WIRELESS SENSOR NETWORK USING BLUETOOTH LOW ENERGY (BLE) FOR CONSTRUCTION NOISE MONITORING," 2015.

[30] “xPico 250 Embedded IoT Gateway.” Accessed: Jan. 07, 2025. [Online]. Available: https://www.lantronix.com/products/xpico-250/
[31] J. Porter, B. Kuzior, J. Garretto, and Y. Sapkota, “YSU Tag - Firmware,” GitHub. Accessed: Oct. 02, 2023. [Online]. Available: https://github.com/YSU-Tag-Dev/ysu-tag-firmware
[32] “BME688 Datasheet,” Bosch Sensortec. Accessed: Sep. 05, 2023. [Online]. Available: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme688-ds000.pdf
[33] “ADXL343 Datasheet,” Analog Devices. Accessed: Sep. 05, 2023. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/adxl343.pdf
[34] W. Niu *et al.*, “Summary of Research Status and Application of MEMS Accelerometers,” *Journal of Computer and Communications*, vol. 06, no. 12, pp. 215–221, 2018, doi: 10.4236/jcc.2018.612021.
[35] F. A. S. Ferreira de Sousa, C. Escriba, E. G. Avina Bravo, V. Brossa, J.-Y. Fourniols, and C. Rossi, “Wearable Pre-Impact Fall Detection System Based on 3D Accelerometer and Subject’s Height,” *IEEE Sens J*, vol. 22, no. 2, pp. 1738–1745, Jan. 2022, doi: 10.1109/JSEN.2021.3131037.
[36] N. A. Prabatama, M. L. Nguyen, P. Hornych, S. Mariani, and J.-M. Laheurte, “Zigbee-Based Wireless Sensor Network of MEMS Accelerometers for Pavement Monitoring,” *Sensors*, vol. 24, no. 19, p. 6487, Oct. 2024, doi: 10.3390/s24196487.
[37] “AD5940/AD5941 Datasheet,” Analog Devices. Accessed: Sep. 05, 2023. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ad5940-5941.pdf
[38] J. Porter, “TagView,” GitHub. Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/YSU-Tag-Dev/tag_view