DEVELOPMENT AND CHARACTERIZATION OF AN IOT NETWORK FOR AGRICULTURAL IMAGING APPLICATIONS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Jacob Wahl

June 2020

COMMITTEE MEMBERSHIP

TITLE:	Development and Characterization of an IoT Network for Agricultural Imaging Applications
AUTHOR:	Jacob Wahl
DATE SUBMITTED:	June 2020
COMMITTEE CHAIR:	Jane Zhang, Ph.D. Professor / Graduate Coordinator Department of Electrical Engineering
COMMITTEE MEMBER:	Vladimir Prodanov, Ph.D. Associate Professor Department of Electrical Engineering
COMMITTEE MEMBER:	Bridget Benson, Ph.D. Associate Professor Department of Electrical Engineering

ABSTRACT

Development and Characterization of an IoT Network for Agricultural Imaging Applications

Jacob Wahl

Smart agriculture is an increasingly popular field in which the technology of wireless sensor networks (WSN) has played a large role. Significant research has been done at Cal Poly and elsewhere to develop a computer vision (CV) and machine learning (ML) pipeline to monitor crops and accurately predict crop yield numbers. By autonomously providing farmers with this data, both time and money are saved. During the past development of a prediction pipeline, the primary focuses were CV and ML processing while a lack of attention was given to the collection of quality image data. This lack of focus in previous research presented itself as incomplete and inefficient processing models. This thesis work attempts to solve this image acquisition problem through the initial development and design of an Internet of Things (IoT) prototype network to collect consistent image data with no human interaction. The system is developed with the goals of being low-power, low-cost, autonomous, and scalable. The proposed IoT network nodes are based on the ESP32 SoC and communicate over-the-air with the gateway node via Bluetooth Low Energy (BLE). In addition to BLE, the gateway node periodically uplinks image data via Wi-Fi to a cloud server to ensure the accessibility of collected data. This research develops all functionality of the network, comprehensively characterizes the power consumption of IoT nodes, and provides battery life estimates for sensor nodes. The sensor node developed consumes a peak current of 150mA in its active state and sleeps at 162µA in its standby state. Node-to-node BLE data transmission throughput of 220kbps and node-to-cloud Wi-Fi data transmission throughput of 709.5kbps is achieved. Sensor node device lifetime is estimated to be 682 days on a 6600mAh LiPo battery while acquiring five images per day. This network can be utilized by any application that requires a wireless sensor network (WSN), high data rates, low power consumption, short range communication, and large amounts of data to be transmitted at low frequency intervals.

Keywords: Internet of Things (IoT), ESP32, Sensor Node, Gateway Node, Bluetooth Low Energy (BLE)

ACKNOWLEDGMENTS

First and foremost, I would like to endlessly thank my parents and sisters for their constant love and for promoting the importance of education.

I would like to thank my advisor, Dr. Zhang, for offering support throughout this process and suggesting this topic to me. Thank you to Dr. Prodanov for recommending the development board used in this thesis and for valuable suggestions throughout my development process. Thank you to Dr. Benson for providing critical and constructive feedback of my paper.

I would like to thank my close friends Avery, Donald, Elysa, Faye, Jennie, Madi, and Sarah for the friendship and encouragement over the past five years. You all truly made Cal Poly a home and a wonderful place to learn, and I cannot thank you enough.

A final thanks goes to my graduate cohort that has persevered through the last five years of school together, always providing support rather than competition.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Background and Overview

1.1.1 Technology in Agriculture

The agricultural industry as a whole is facing significant challenges, from the rising costs of supplies, labor shortages, a shift in consumer preferences for sustainability, and the surge in global population. These challenges have led to more innovation and investment capital put into modern agriculture than ever before [1]. As a result, the use of different complex and connected technologies have been employed to assist the industry enhance operational efficiency, improve productivity, and reduce costs. Primarily, the use of Internet of Things (IoT), Computer Vision (CV), and Machine Learning (ML) are being used together to develop intricate pipelines to keep farmers knowledgeable about their farms, crops, and yields.

One specific application of using the combination of IoT, CV, and ML is to build a remote crop monitoring and yield prediction system. The crop monitoring system would provide farmers with easy access to valuable information regarding their plants' growth conditions, and thus support farmers' decisions about crop production. Additionally, yield predictions prior to harvest dates is essential for improving product management and marketing plans. It would help farmers monitor their fields throughout the year and allow flexibility in hiring an appropriate number of laborers when they will be needed. Misestimating harvest times and labor requirements result in large fiscal losses as well as crop waste in the field due to untimely harvests. A system as described is incredibly advantageous for farmers.

1.1.2 Motivation

Significant work has already been done in the computer vision and machine learning sections of the described pipeline, leaving IoT open to be explored [2], [3]. This section addresses issues brought up in two previous Cal Poly thesis works focused on CV and ML that became the motivation behind this paper to develop an IoT system.

The first work, *Towards a Strawberry Harvest Prediction System Using Computer Vision and Pattern Recognition*, was written by Andreas Apitz and completed in June 2018. He attempted to develop a data acquisition, computer vision, and prediction pipeline to reliably predict the data and yield of the field under

test. Apitz focused on many different CV techniques to detect and segment red strawberries (the crop of interest) out of images he had collected in Cal Poly's crop fields. He tested thresholding in various color spaces, obstacle masking, and Hough transforms to accomplish this. Apitz also explored three different methods for data collection: manual handheld camera, mounted camera, and quadcopter drones. For his work, he primarily used the manual and drone method for his testing, though he noted that the mounted camera method could be the best option (but failed to test it as it required extensive design). From these methods, he recorded many shortcomings in image collection. The manual handheld camera method was time consuming and incredibly inconsistent. Changes in viewing angles, illumination levels, and acquisition frequencies (ranging from once a day to once a week) all contributed to more difficult computer vision development. The drone method was quick but was unable to capture images closer than about 10 meters over the ground using autopilot, and this altitude made it impossible to detect strawberries accurately and reliably. This meant to detect strawberries, the drone had to be piloted manually. Even with this factor, the drone would have to fly at least 3 meters above the ground, making it difficult still to extract strawberries from images. A side-by-side view of the handheld camera and piloted drone data acquisition methods are shown below in Figure 1.

Figure 1: Handheld Smartphone Image (left) & Drone Image at 3m (right) [2]

As mentioned in the previous paragraph, Apitz proposed a mounted camera option and chose this as the best data acquisition method based on a subjective decision matrix. The matrix considered cost, power, proximity, ease of use, image quality, time required consistency, and exposure as performance metrics. The mounted camera idea involved mounting a series of cameras through the field to "automatically and

periodically take pictures of the plants and upload them" to a remote server, through "Bluetooth or Wi-Fi." Though not stated specifically, Apitz was suggesting an IoT network structure of devices.

The second work, *Strawberry Detection Under Various Harvestation Stages*, was written by Yavisht Fitter and completed in March 2019. Fitter analyzed various techniques, both computer vision and machine learning, to detect strawberries at various stages in their growth cycle. He implemented histogram of oriented gradients (HOG), local binary patterns (LBP) and convolutional neural networks (CNN) on a limited custom-built dataset. His data collection method consisted of using a smartphone to take images "twice a week … at an average height between 4 to 5 feet" [3]. He collected a total of 600 images and used 75% to train and 25% to test his methods. While the testing and development of his three techniques proved to be useful, Fitter stated that they could be improved, and other methods were not attempted due to a lack of data.

The common theme within both author's recommendations is the importance of good image collection needed for complex processing down the line. Some of the problems they encountered with their collection methods include inconsistencies of acquisition frequency, varying illumination levels, varying viewing angles and heights, as well as a time-consuming collection process. This serves as the primary motivation behind this thesis work, to solve these problems and consider the suggestions of these authors.

1.1.3 Significance of Quality Data

Computer vision is the science of acquiring, processing, and analyzing digital images from the real-world to extract information for decision making. Machine learning provides systems the ability to automatically learn and improve from experience to make predictions or extrapolations about data. Both fields completely rely on sample data to perform their functions. The quality, quantity, and consistency of image data is incredibly significant for the complex processing methods found in computer vision and machine learning models. Larger datasets used in computer vision and machine learning processes can help learn model parameters and improve optimization calculations [4]. This significance is why it is useful to set out to solve the data collection failures documented in previous works. The CV and ML processes must become more robust, efficient, and reliable to implement an accurate crop detection and yield prediction system.

1.2 Statement of Problem

There is an unreliable source of image data to be properly used within the complex CV and ML processes that have been developed to predict harvest yields. The motivation of this thesis is to address the issues described by previous thesis works and to improve the image collection process in order to create a more robust and useful crop detection and yield prediction model. Solving this problem could assist in giving farmers an autonomous method of analyzing their crops, reducing time and cost to them, and to reduce crop waste in the field due to untimely harvests. This thesis proposes the design and development of a custom IoT system to be used as the sole data collection step in a crop detection and yield prediction pipeline that addresses and solves the deficiencies of methods used previously.

1.3 IoT Background

The Internet of Things technology is such a broad and commonly used term. This section attempts to briefly narrow the scope of IoT to this paper and to give enough background to understand basic terms and systems.

IoT describes a movement towards web-connected devices used to make common or simple tasks automated. In our case, it describes a network of wireless sensors and devices that collect and share data to the internet to be utilized elsewhere. Image and analog sensor data could be used as inputs to a computer vision process or to a machine learning model, for example. Typically, an IoT ecosystem consists of four major components: IoT devices, communication technology, the internet, and data storage [1]. Figure 2 below shows the basics of a connected network. The key advantage to an IoT system is communication in remote and inaccessible areas, making it an emerging technology for farms across the world. According to a 2016 Machina Research report, the number of IoT connected agricultural devices is expected to grow from 13 million at the end of 2014 to over 225 million in 2024, an increase of 1,785% [5].

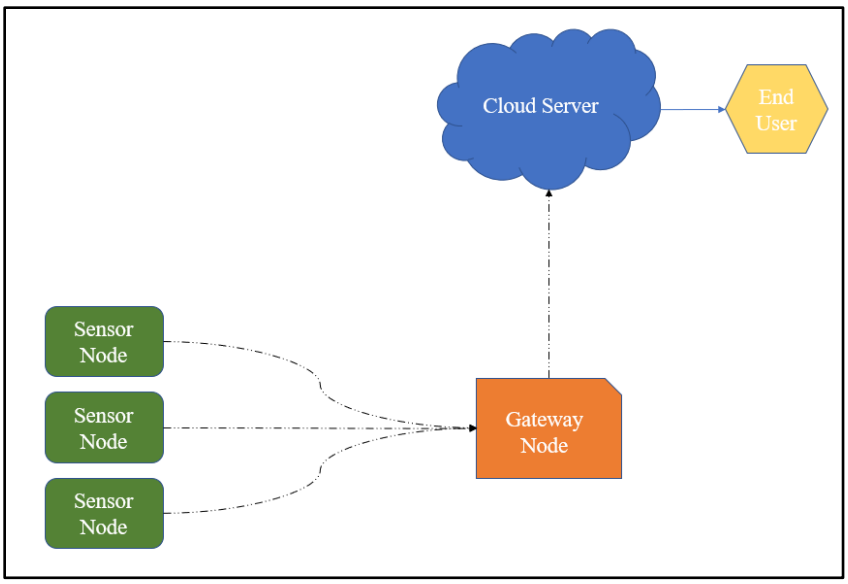

Figure 2: Typical IoT Network Structure

The general flow of data is: (1) data collection by Sensor Nodes, ranging from temperature data to image data, (2) data transmission over an RF communication link from Sensor Nodes to Gateway Nodes, (3) Gateway Nodes store data and upload previously collected data to Cloud Servers via the internet, and (4) easily accessible storage of data in the Cloud Server to be utilized by the end user from anywhere in the world with an internet connection.

As a summary of important parts of an IoT network, the following list defines key terms.

- Sensor Node – Performs data collection and storage, has ability to receive and transmit data over RF communication
- Gateway Node – Ability to receive and transmit data over RF communication, the internet access point, uploads received data via internet
- Cloud Server – Data storage on the internet, easily accessible via the internet from anywhere, at any time for the end user

1.4 Scope of Work

This thesis provides design and development decisions in creating a low-cost, low-power IoT system to be used in agricultural applications. This research will aim to build a standalone prototype system with the ability to be expanded upon. The main goal for the prototype system is to develop essential functionalities of a sensor node, a gateway node, a cloud storage site, and the communication links between

all three. As the prototype sensor node will be designed to monitor crops, the device lifetime should be at least 1-year of uninterrupted operation on battery power. Each node in the network should cost no more than $30 to enable affordable implementation for farmers. The end user of the image data should be able to access captured pictures at any time via the internet.

Chapter 2 reviews relevant research papers pertaining to similar IoT systems developed. Chapter 3 reviews a preliminary system design for this thesis while providing specific goals and hardware case studies. Chapter 4 overviews the hardware system chosen for this thesis, the ArduCAM IoTai, and details features of the development board as well the core SoC, the ESP32. Chapter 5 documents design decisions and processes while developing key features of the IoT system. Chapter 6 details a comprehensive characterization of the system including timing constraints, current consumption, battery life estimates, and a performance evaluation. Chapter 7 concludes the thesis and provides a functional summary while suggesting future work to be done. The appendix includes references to all code developed for this thesis work.

Chapter 2

LITERATURE REVIEW

This chapter discusses three previous papers or projects that have developed IoT systems in agricultural settings. The goal of this literature and project review is to gain an understanding of what designs and decisions were made to develop successful similar systems in the past and learn from them. While none of the works reviewed here tackled the specific task of capturing and connecting image data to the internet, they do contain valuable information regarding wireless communication and unique hardware solutions. In fact, the task of agricultural IoT imaging for the purpose of crop detection and yield prediction (albeit a very narrow topic) seems to have never been attempted before.

2.1 Vinduino

2.1.1 Overview

The first project was born out of necessity by a California farmer. Vinduino (Vineyard + Arduino), was created to better manage the irrigation system of a southern California vineyard. The primary goal of the system was to provide a low-cost and easy to build system with rugged components for optimizing agricultural irrigation.

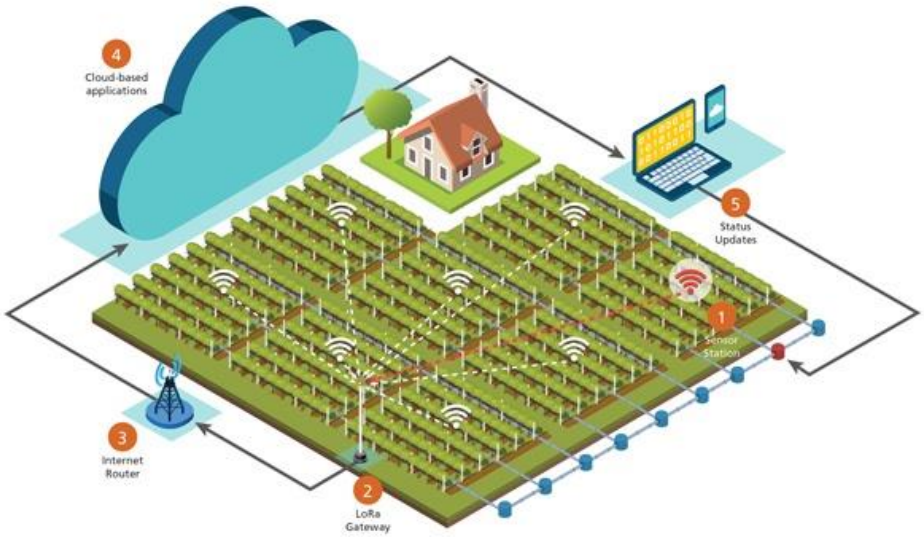

Figure 3: Vinduino High Level Network Flow [6]

The process of the system developed is depicted in the Figure 3. The sensor nodes are placed out in the fields where they can collect any needed data from crops. They then send the collected data out to the gateway node, placed near the edge of the field and close to an internet router. This gateway can then upload the data via the router to a cloud server, where the data is aggregated into graphs to be easily analyzed by the farmer. After checking the data, the farmer can then send out simple commands to the sensor nodes to change irrigation control, if needed.

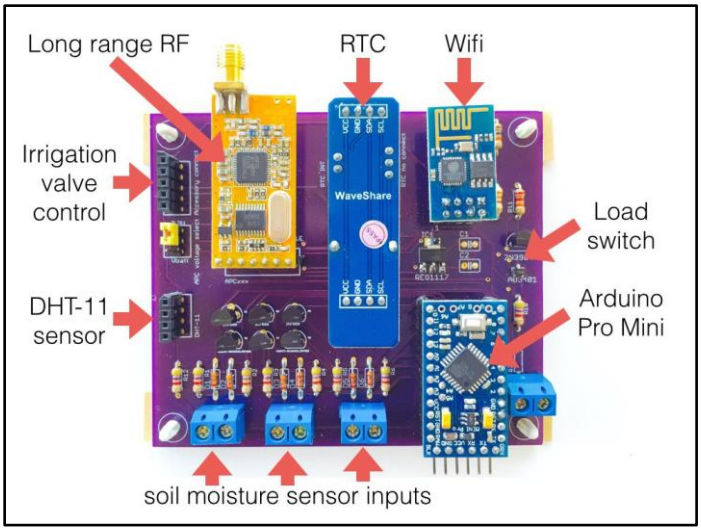

Figure 4: Vinduino Remote Sensor Node [6]

The project uses multiple soil moisture sensors, located at different depths to prevent overwatering and control irrigation to not exceed the active root zone. The solar powered remote sensor nodes have three gypsum soil moisture sensors and several options for temperature or humidity sensors. The designed board, shown in Figure 4, includes a Globalsat LM-210 LoRa module for long-range wireless communication (up to 6 miles), a built-in solar battery charger, and a built-in real time clock (RTC) for precise irrigation timing. This board also has the option to plug in an ESP8266 SoC to provide Wi-Fi connectivity. The functionality of the sensor node is controlled by an Arduino Pro Mini [6].

The gateway node is based on LoRa (long-range) communication. It collects data that is sent by all the sensor nodes and is able to transmit that to a cloud server via a nearby internet router. The gateway and its connection to the internet is separate, adding one more (possibly unnecessary) step in the data flow. The internet router pipes the data to a ThingSpeak cloud server, a simple analytics platform for IoT projects.

This platform service is made by MathWorks, the makers of MATLAB. ThingSpeak allows data to be aggregated, visualized, and analyzed in real-time in the cloud, shown in Figure 5.

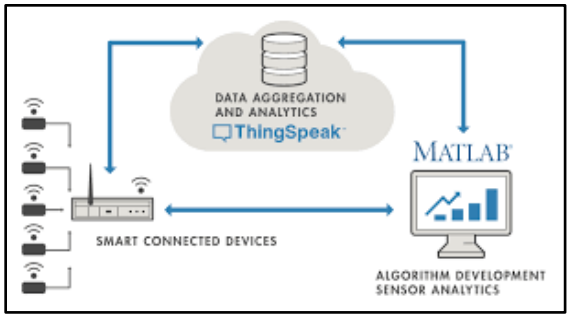

Figure 5: ThingSpeak (by MathWorks) Analytics Platform

Any data that has been uploaded to ThingSpeak can be processed and analyzed in MATLAB. The farmer can view this data from wherever he wants with an internet connection on his own personal machine. With these visualizations, he can make decisions about irrigation and send simple commands to the irrigation valves via the sensor nodes to adapt to any changes.

2.1.2 Analysis

While this previous work was not an academic paper, it provides an incredible amount of information and documentation on its development, which is why I chose to include it in this thesis. This project saw real-world success and was completely implemented in this farmer's vineyard. The success of this project's functionality should not shield it from a critical review and analysis, though. This section will briefly overview what this project did well, and what could possibly be improved.

The data being collected and processed was primarily soil moisture and temperature data. This translates to a small packet size of actual data being transmitted at a time, usually with a maximum size of ten bytes. This lends itself nicely to the low bandwidth LoRa communication method. While this is discussed in more detail in Chapter 3, LoRa is able to travel large distances because of a cut to its data rate, making transmission of a small amount of data allowable. This small amount of data is easily handled by the ThingSpeak platform, which is able to be used free of charge with certain data limitations. These limitations include a yearly message amount of three million "messages." A "message" is defined as a write of up to 8 fields of data to a ThingSpeak channel. This limitation is ideally used for data with specific fields, like temperature or moisture, which is typical for many IoT applications. The decisions to use LoRa

and ThingSpeak are intentional and gave me some insight into design tradeoffs including data rate, transmission distance, data size, and cloud platform limitations.

The sensor and gateway nodes in this project were designed and implemented with discrete modules on to a custom printed circuit board. In my opinion, the design of the sensor node has a redundancy that could be fixed, and the physical hardware placement could be given a second look. On the sensor node, the optional ESP8266 SoC is used only as a Wi-Fi replacement if the LoRa module was deemed unnecessary. By making the ESP8266 a permanent installment of the node and removing the Arduino Pro Mini, the node would become a lot easier to modify and just as easy to control. The SoC does indeed include native Wi-Fi, but also integrates a 32-bit Tensilica microprocessor. This processor has many more capabilities than an Arduino Atmega328 microcontroller, including lower power consumption, more peripheral interfaces, and a lower cost. You can even program and upload code to the ESP8266 from the Arduino IDE. In addition to this hardware replacement, an improvement to the physical layout of the system can be made. Looking at Figure 3 of the system flow, it would be advantageous to move the location of the gateway node to near the farmer's house. The farmer can easily install an internet router in his home (if not installed already) and completely cut out the internet router on the opposite side of the field. This would reduce cost and time to the farmer.

2.2 Long-range & Self-powered IoT Devices for Agriculture & Aquaponics Based on Multi-hop Topology

2.2.1 Overview

This paper presented at the 2019 IEEE 5th World Forum on Internet of Things (WF-IoT) presents a prototype design of a long-range, self-powered IoT device for use in precision agriculture [7]. The IoT system presented is shown in Figure 6.

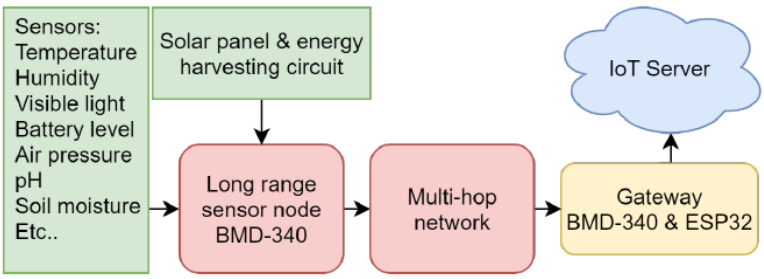

Figure 6: IoT System Architecture [7]

The system is designed to collect temperature, humidity, light, air pressure, soil acidity, and soil moisture data and transmit it back to an IoT server. The sensor nodes are designed with BMD-340 modules based on the nRF52480 microcontroller with an on-board antenna. The controller has Bluetooth 5 long-range support integrated into the chip. They use the bq25570 integrated circuit from Texas Instruments for energy harvesting, battery charging, and voltage conditioning. The nodes are powered from a Li-Ion battery with a 120mAh capacity. The sensor nodes are programmed to transmit at 125 kilobits per second in long-range mode using forward error correction (FEC) scheme to perform error detection and correction on received data.

To transmit this data collected at the sensor nodes to the gateway node, based on the ESP32 SoC, a custom developed “multi-hop” network is used. This network allows the system to extend its range to full coverage of large crop fields. The protocol enabled data transmission distances of 1.8km per hop, extending sensor coverage of almost any size to be viable. As this system will have many sensor nodes and a few gateway nodes placed around a large field, this network topology is able to route messages towards a dedicated gateway, where the closest gateway is chosen after the network has adapted. Once data gets to a gateway node, the messages are published via MQTT over Wi-Fi to a local network server.

2.2.2 Analysis

This paper focuses on the multi-hop network protocol for a large-scale agricultural IoT system to produce a low-power architecture. Though the scale of the described system is much larger than what might be implemented by this thesis, it is useful to see what large scale systems need to function efficiently. It was useful to read about the multi-hop protocol that extends the range of the sensor nodes to an entire crop field as well as making it use as little power as possible. This data transmission organization is likely the biggest obstacle to creating an efficient and scalable IoT system. Some thought should be put into creating a similar topology for the system described in this thesis work, even though it is a simpler system. The paper also included a brief discussion of power consumption of a sensor node and recorded a sleeping mode power of 6.94 microwatts [7].

2.3 Smart Agriculture Farming with Image Capturing Module

2.3.1 Overview

This research paper, submitted to the 2019 Global Conference for Advancement in Technology (GCAT), focuses on the development of building an IoT system with a camera module device to analyze possible diseased crops [8]. The scope of this paper is similar to this one but aims to solve a different agricultural problem.

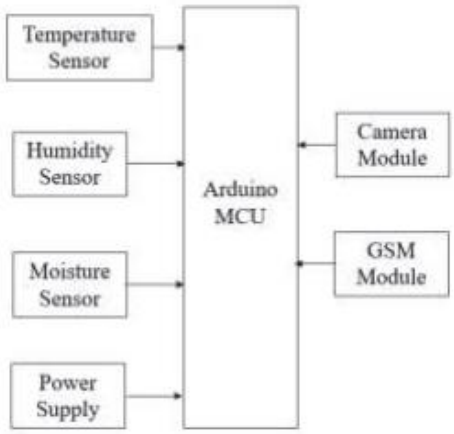

Figure 7: Block Diagram of IoT System Implementation [8]

The architecture of the system is given in Figure 7. An Arduino Mega 2560 acts as the central processor and initiates all tasks in the process. The model uses temperature, humidity, and water moisture sensors connected to the analog pins of the Arduino as well as an OV7670 0.3-megapixel camera. Data collected by the system is received by the Arduino and sent to the cloud using an ESP8266 as a Wi-Fi module.

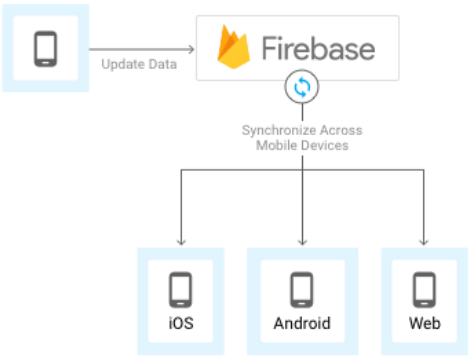

Figure 8: Firebase Utilities

The team used Firebase (backed by Google) as cloud storage for all data collected by the device. Firebase is a user-friendly platform to create and use databases for dynamic storage. The data can then be accessed by anyone with the proper authentication from any internet connected device. Figure 8 shows the synchronization of data over any connected device.

2.3.2 Analysis

This research demonstrates that it is possible to use an IoT network for image transfers and cloud storage. The paper shows that low-cost hardware and free software can be used to create a simple but effective IoT system. In the conclusion, the researchers state that the image capturing module functions slowly due the Arduino. Unfortunately, the paper does not discuss power requirements as the sensor node seems to run on a continuous power supply. The focus of the research was proof of concept and did not result in any optimization of the functionality. Despite this fact, the discussion on the free cloud storage used to store both sensor data and image data was helpful. Further research into various cloud storage platforms must be made for my system and Firebase seems like a promising choice.

Chapter 3

PRELIMINARY SYSTEM DESIGN

3.1 System Goals

The proposed system will be the sole data collection step in this CV and ML pipeline and focus on creating a base functionality design that can be its own standalone point-to-point system able to monitor the growth of a single viewable area in a farm. The goal of this thesis is to develop a low-cost, low-power image acquisition IoT solution that addresses the common issues seen by previous works.

In order to create a good solution, the development focused on the following areas.

- Low Cost: The designed system should be affordable for the end user and consumer. Each node should be at most $30.
- Low Power: The system (especially the sensor node) should use as little power as possible. The nodes must have long lifetimes while running on battery power. The longer the devices can run on a single battery charge, the less frequent they have to be maintained by a human. The sensor nodes should last at least 1-year operating on battery power.
- Scalability: While this thesis aims to develop a single sensor node and gateway node pair with accompanying significant functionality, this network will likely have to expand in size to meet farmer demands. A focus should be put on scale so that the network is easily able to accommodate change and improvement down the line, such as a mesh network.
- Autonomy: This system should be primarily autonomous to reduce the amount of effort by humans to collect and utilize this image data. Once this system is in place, no human input or maintenance will be needed until the battery is discharged.

Figure 9 displays a preliminary system design before any decisions were made on hardware or wireless communications. This follows the typical IoT network structure described earlier.

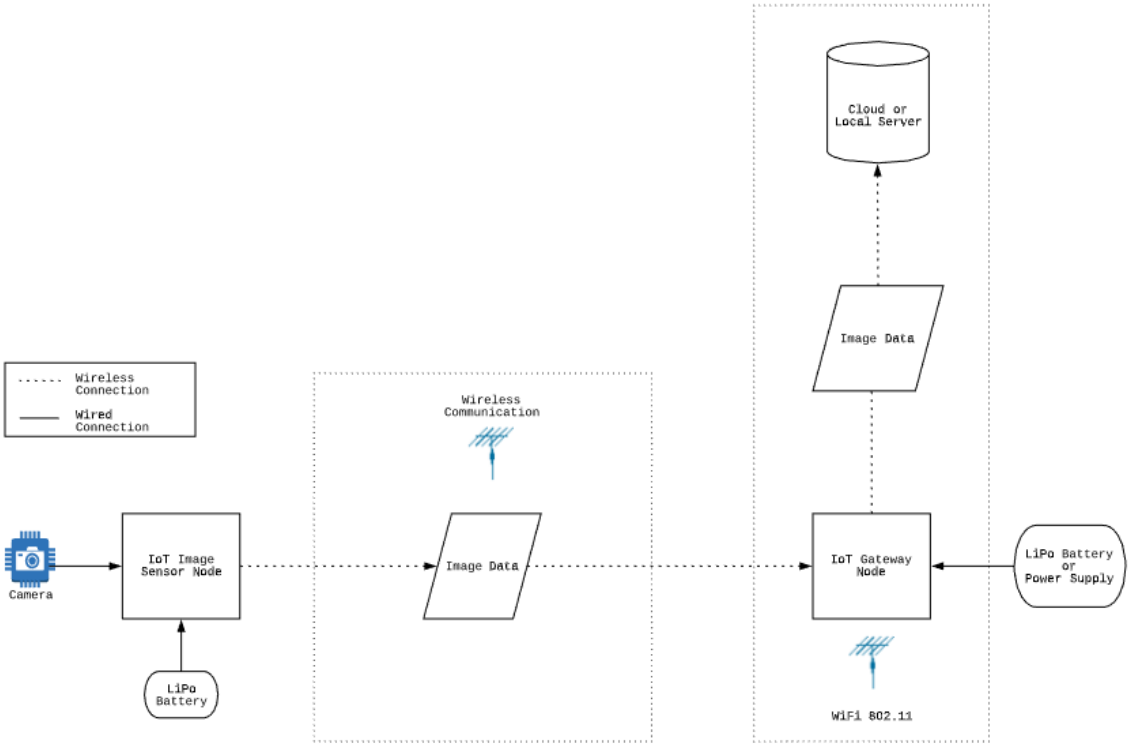

Figure 9: Preliminary High-Level System Design

The decisions now must be made on what hardware will be used as the sensor and gateway nodes, which wireless communication technology will be used to link the nodes, and the best peripherals to be connected before we start functional development.

The two most significant factors to this design are the hardware/software ecosystem and the wireless communication technologies chosen. These two things will facilitate data flow and data speeds throughout the system. The following two sections will describe the needs of the system with respect to each factor and compare different technologies reviewed.

3.2 Wireless Communication

Choosing the best communication method for sensor to gateway node links as well as the gateway link to the internet is significant. IoT systems generally have three different categories of wireless communication technologies: personal area network (PAN), Cellular, and low-power wide area networks (LPWAN) [9]. Two different PAN technologies and two different LPWAN technologies are reviewed in the section. Many factors were reviewed in this research including data rates, power consumption, transmission distances, popularity, and scalability.

3.2.1 LPWAN: LoRa and SigFox

The LoRa (Long Range) communication technology is a commonly used IoT protocol. This technology was the driving link for the Vinduino project, described in Chapter 2. LoRa is a low-power wide-area network technology. It is based on spread spectrum modulation techniques and uses license-free sub-gigahertz radio frequency bands like 433MHz or 915MHz. This enables very long-range transmissions with low power consumption. While LoRa has a line-of-sight range of multiple kilometers, it has incredibly low bandwidth. This puts significant limitations on the data that is can be sent. The message payload needs to be as small as possible and encoded as binary data. Most recommendations keep the payload under 12 bytes. Because the data has to be transmitted over such a long distance, the interval between messages should be in the range of several minutes [10]–[13].

SigFox is a very similar communication technology as LoRa. The technology operates in the license-free frequency bands around 920MHz. It is a lightweight protocol with very little overhead needed to transmit data which means it consumes very little power. SigFox only supports one-way communication as opposed to the bidirectionality of LoRa. The pitfalls of SigFox follow those of LoRa. The communication technology is ultra-narrowband and will only support 140 messages per day per device with 12 byte payloads [14], [15].

While both LPWAN technologies have their advantages in most classic IoT networks, their biggest limitation is the largest disadvantage for our system. In order to create low-power sensor nodes for this imaging system, the transmission time must be minimized. To do this, the communication bandwidth must be large, and the data rate must be fast enough to transmit kilobyte sized images in a reasonable amount of time.

3.2.2 PAN: ZigBee and Bluetooth Low Energy

ZigBee is a mesh network protocol designed to carry small packets of data over short distances while maintaining low power consumption. Mesh networks operate on the principal that the information from a single sensor node travels on a web of interconnected nodes until the transmission finds its way to the gateway. It uses a version o the IEEE 802.15.4 standard and is widely used in local area sensor networks. It has a short range of approximately 10 to 100 meters with a 100 milliwatt transmit power [16], [17].

The wireless standard known as Bluetooth should sound familiar to most people. It describes a short-range wireless communication technology that has penetrated the consumer market faster than any other. The Bluetooth Special Interest Group introduced Bluetooth Low Energy, a newer standard than the classic serial port profile (SPP) Bluetooth, and this now has monopoly over the consumer electronic short-range communication market. It is used in almost every device people use on a day to day basis and has become synonymous with in-home IoT solutions. BLE has a slightly shorter range than ZigBee, but also has a much higher data rate at around 1 megabit per second for short bursts. The low energy aspects come when the device sleeps in between those bursts, which requires less power usage. ZigBee does not have this functionality [16], [17]. BLE also has the ability to be configured in a mesh or star network topology, giving it scalability for variable size networks.

As the idea for a large IoT network has potential for the system being developed, these PAN technologies are an attractive option. The number of sensor nodes will most likely be large and constrained to a relatively small area, so the distance between these nodes will be small. This enables the use of such communication technologies.

3.2.3 Summary of Wireless Communication Technologies

Table 1 describes the four communication technologies discussed while comparing certain key characteristics between them.

Table 1: Wireless IoT Communications Comparison

	BLE (v4)	**ZigBee**	**LoRa**	**SigFox**
Range	< 100m	100m	> 1km	> 1km
Topology Options	Point-to-point, Mesh, Star	Point-to-point, Mesh	Line-of-Sight	Line-of-Sight
Data Rate	1Mbps	250 kbps	27 kbps	600 bps
Power Consumption Rank	1 (Best)	2	4 (Worst)	3
Frequency	2.4GHz	2.4GHz	433 or 915MHz	920MHz

Based on the characteristics of the table shown and the goals of the system to be developed, the best option for wireless communication between the sensor nodes and the gateway is Bluetooth Low Energy. This protocol has the lowest power consumption, the highest theoretical data rate, and the most flexibility in network topology. These characteristics make BLE a great option for our system.

3.3 Hardware Case Studies

This section briefly outlines the core hardware that was assessed to develop this IoT system. Hardware solutions were researched after deciding that BLE would be the wireless communication to transmit data from sensors to gateway nodes. Key characteristics in attractive hardware were various low-power modes, integrated BLE modules, numerous peripheral selections, and cost.

3.3.1 Arduino Nano 33 BLE

The Arduino Nano 33 BLE features a powerful nRF52840 32-bit ARM Cortex M4 processor that runs at 64 MHz. The board has 1 MB of program memory and 256 kB of SRAM [18]. Figure 10 shows the physical development board.

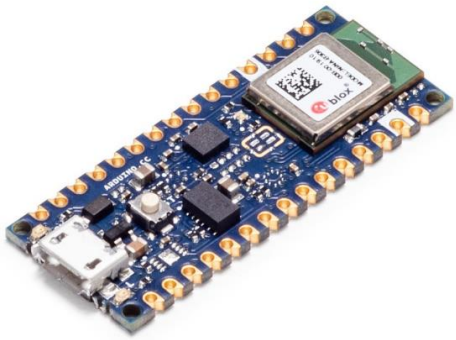

Figure 10: Arduino Nano 33 BLE Board [18]

The board also integrates 14 digital GPIO pins and one UART, SPI, and I^2C interface each. The nRF52840 has a fully integrated Bluetooth 5 module with BLE mode. This chip is also able to run in multiple sleeping modes for low-power consumption in the microamp range.

3.3.2 Particle Argon

The Particle Argon development board is built specifically to be either a sensor or gateway node in an IoT network. The board integrates both and ESP32 SoC as a Wi-Fi coprocessor and the nRF52840 SoC as a Bluetooth module. The physical board is shown in Figure 11.

Figure 11: Particle Argon Development Board [19]

It has plenty of storage with 1 MB of program memory and 256 kB of RAM for the nRF52840 and 4 MB of program memory for the ESP32. There are 20 mixed signal GPIO lines with UART, I^2C, and SPI interfaces. This system has only one PCB antenna (for BLE) but has two separate U.FL ports to attach external antennas for Wi-Fi and BLE [19].

3.3.3 Espressif ESP32 SoC

The ESP32 is advertised as a low-cost, low-power SoC with integrated Wi-Fi and Bluetooth 4.2 capabilities. At the core, there is a dual-core Tensilica Xtensa LX6 microprocessor that runs up to 240MHz. The ESP32 also integrates an ultra-low power co-processor and an RTC for low current consumption with five different sleep modes [20]. Figure 12 shows the surface mount chip that is soldered on to a variety of development boards.

Figure 12: Espressif ESP32 SoC

The ESP32 SoC is available as the core of many different development boards as it is widely used by IoT hobbyists. One of these is of particular interest, the ArduCAM IoTai. This development board ships with an OV2640 2-megapixel camera sensor with an on-board parallel interface, an on-board microSD card reader, and a battery pin connector [21].

3.3.4 Hardware Summary

Table 2 summarizes key points in the hardware solutions reviewed. The characteristics chosen are of importance to the specific IoT system being developed in this thesis.

Table 2: Hardware Features Summary

	Arduino Nano 33 BLE	**Particle Argon**	**ArduCAM IoTai ESP32**
SoC	nRF52840	nRF52840, ESP32	ESP32
Program Memory	1 MB	1 MB, 4MB	4 MB
RAM	256 kB	256 kB	4 MB
GPIO	14	20	24
Wireless Comm.	BT 5.0	BT 5.0, Wi-Fi	BT v4.2, Wi-Fi
On-Board Camera?	No	No	Yes
On-Board Expandable Storage?	No	No	Yes
PCB Antenna?	No	Yes (only for BLE)	Yes
Battery Connector?	No	Yes	Yes
Cost	$20.20	$27.00	$19.99

While all of these platforms have the potential to develop into an IoT system, the cost of the platform is a major selling point. The Arduino and Argon board are feature-rich solutions but would require external purchasing of cameras and expandable storage hardware to be used, adding to the prices listed. The ArduCAM IoTai based on the ESP32 provides a wide array of scalable features and integrates all the wanted peripherals onto a single development board for under $20. For these reasons, the ESP32 based ArduCAM IoTai is chosen to be at the heart of the proposed IoT system.

3.4 Final Design Summary

Using these, a point-to-point system will be developed to produce the functionality of an IoT system. The system will have the goal of being implemented in a crop field as depicted in the following figure. The sensor node and gateway node will both be based on the ArduCAM IoTai ESP32 but will physically be two separate devices. The two nodes will communicate with each other using the Bluetooth Low Energy (BLE) standard. The gateway will upload aggregated image data to a cloud server over Wi-Fi. The sensor node will be powered by a LiPo battery as it will be deployed in a field far away from a constant power source.

Since the gateway node will be assumed to be near a stable internet connection, it will also be assumed it will be near a constant power source. This description is illustrated in Figure 13 below.

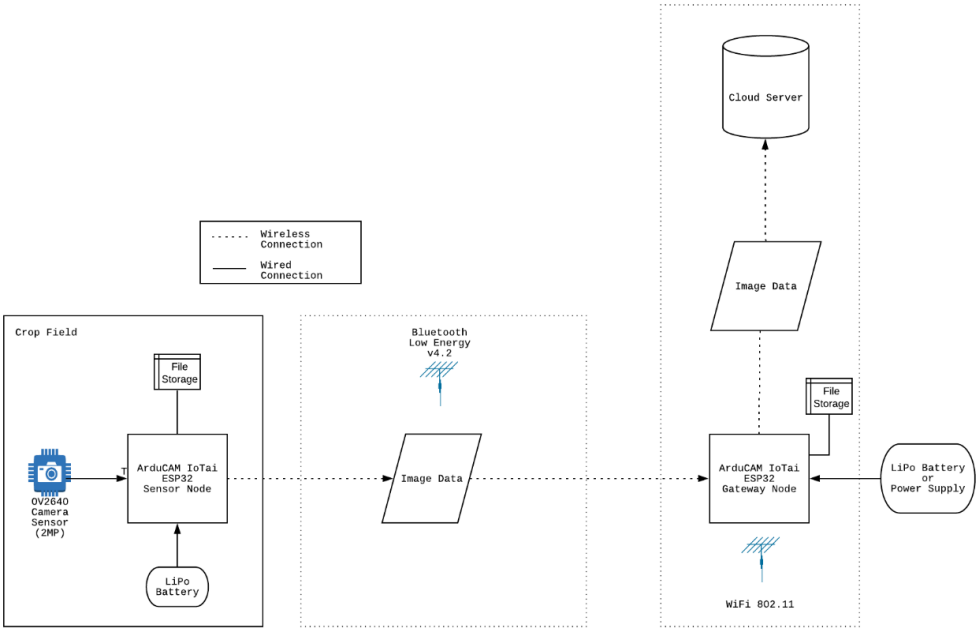

Figure 13: Final High Level IoT System Design

Chapter 4

HARDWARE OVERVIEW

This chapter will give an overview of the hardware chosen for the development of this thesis. A detailed discussion of the features of the ArduCAM development board, ESP32 system on a chip (SoC), and OV2640 image sensor module will be included, as these are the primary pieces of the IoT system being developed.

4.1 ArduCAM IoTai Development Board

The device chosen to function as both the sensor node and gateway node in our network is the ArduCAM IoTai UNO PSRAM development board, shown in Figure 14. This hardware module is an all-in-one solution with many attractive features for developing a custom IoT system.

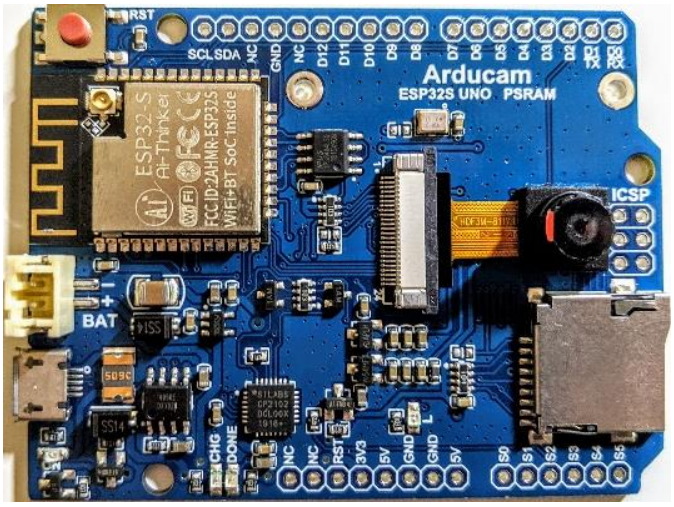

Figure 14: ArduCAM IoTai Development Board

This development board is based on the ESP32 SoC and the OV2640 2 Megapixel Camera Module, which are both described later in this chapter. Apart from these two components, the development board also features the following useful parts:

- MicroSD Card Slot
- LiPo Batter Connector (2-pin)
- 4MB PSRAM, 32Mbit Flash
- USB-Serial Interface
- PCB Antenna for RF Communication + U.FL Connector for External Antenna

The microSD card slot and plentiful on-board flash storage are large factors in making this system a scalable network. Having the option to be able to variably expand the on-board storage of this board allows for greater flexibility in its applications. MicroSD cards are sold with a wide range of storage sizes, from 64MB to 512GB, depending on the type of card. The flash storage chips also give a way to organize image files well due to the fact that a basic tree file system can be implemented on them rather easily. Because our application will be based around the storage of images that will vary in size, it is important to have plenty of storage space on these devices as well as a good organization system to easily access the image files in software.

The development board features a 2-pin battery connector to power the system. The sensor node of the proposed system will be far away from any power sources, so it must be powered by a battery. This connector leads to a voltage regulator that will power the ESP32 SoC at 3.3 volts, therefore, we can have batteries of different nominal voltages plugged in to our development board, expanding our options.

Program memory is incredibly important in this design. Apart from storing images, the board must be able to store firmware and its overhead. With 32Mbits of flash memory, the board can store a respectable amount of firmware, and with 4MB of PSRAM (pseudo-static random-access memory) it can create and manipulate significant amounts of program variables when firmware runs. For a comparison, an Arduino UNO based on the ATmega328 chip has 32kB of flash memory and 2kB of SRAM (static RAM) memory. Our system will need a significant amount of memory for programs and variables as the system will be using Wi-Fi and Bluetooth technologies which include large overhead sizes for controllers and other needed data.

Though it is not necessarily a requirement for programming, the board features a USB-serial chip that makes program compilation and uploading very simple. The alternative to this would be to use an FTDI adapter which allows for many common problems to occur when trying to program the board, like improper wiring or compatibility issues.

Lastly, the development board includes a PCB antenna for radio frequency (RF) transmissions from the ESP32, as well as a U.FL connector for the option to attach an external antenna. This feature is important as the system uses both Wi-Fi and Bluetooth technologies for the transmission of image data and this allows for flexibility in supplementing the transmission output power of the device.

4.2 Espressif ESP32 SoC

The most important part of the development board is the SoC it is built around, the brains of the system. The ESP32, created by Espressif Systems, is a low-cost, low-power system with both Wi-Fi and dual-mode Bluetooth capabilities. At its heart, the chip features a dual-core Tensilica Xtensa LX6 microprocessor with a clock frequency of 240MHz. Designed for mobile devices and IoT applications, the ESP32 achieves ultra-low power consumption through multiple power saving features and relying on its real-time clock (RTC) and ultra-low power co-processor (ULP). The full functional block diagram is shown in Figure 15.

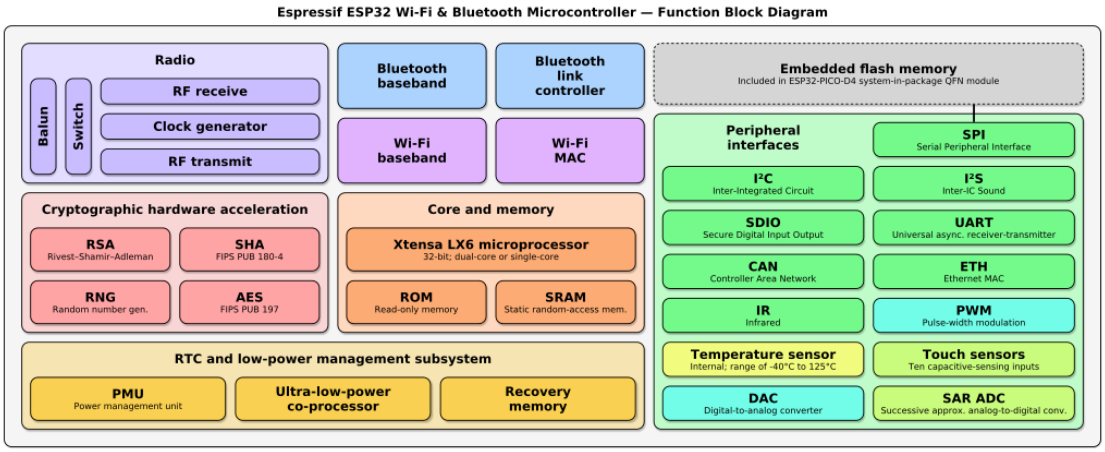

Figure 15: ESP32 High Level Functional Block Diagram [20]

The ESP32 also features many interfaces for the addition of peripheral devices. These include standard I^2C (inter-integrated circuit), DAC (digital-to-analog converter), SPI (serial peripheral interface), and SDIO (secure digital input output). All of these interfaces are incredibly important for designing a scalable IoT system that can adapt to changes and additions. Though this thesis will not include any external devices that need to be directly interfaced with (like analog sensors), having the option and bandwidth to do so is a large advantage for this system.

A more detailed look into the electrical and radio characteristics of the functionalities on this chip will be discussed later in the Chapter 5, Development. These characteristics and specifications will be addressed as they are needed in the development process outlined in that chapter.

4.3 OV2640 Image Sensor

This system being developed relies on images, so the camera module used is of great importance. The module must be able to capture quality images and have the ability to control many image processing functions like white-balancing, saturation, de-noising, and similar processes.

The OV2640 camera module is a low voltage CMOS image sensor that provides full functionality of a single-chip camera and image processor in a small package. This module is included with purchase of the ArduCAM development board, though it is possible to replace this camera with another parallel interface image sensor if necessary. The physical module is depicted in Figure 16.

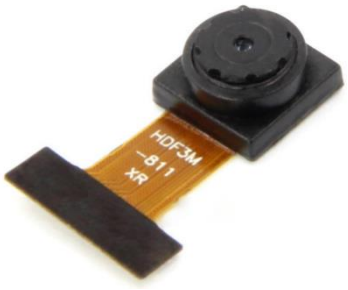

Figure 16: OV2640 CMOS Image Sensor Module [22]

This 2-megapixel camera module is capable of UXGA (ultra-extended graphics display) image display mode where the resolution of a captured image is 1600 pixels horizontally by 1200 pixels vertically (1600x1200). This totals 1.92 million pixels on the display. For our agricultural imaging application, this image quality is sufficient. If better image quality is needed at later time, a higher quality parallel interface camera module can be purchased separately and used with this system.

A very important aspect of this module is the fact that it features an on-chip compression engine. This engine consists of three major blocks: a DCT (discrete cosine transform) block, a quantization block, and an entropy encoding block (Huffman encoding) [22]. These three blocks perform JPEG compression on images captured, greatly reducing the size needed to represent and store the image while keeping the image quality almost unchanged. Reducing the size of these images will decrease the amount of time it takes to transmit the image data, therefore decreasing the amount of power being used by the system. This will allow for a longer device lifetime while running on battery power.

Another significant feature of this camera is the ability to control so many image control functions automatically and manually. These automatic control functions include exposure control, gain control,

white-balance, and black-level calibration. Exposure is the amount of time the CMOS sensor array is exposed to light; gain controls the amplifier gain following sample and hold circuitry; white-balance control removes unrealistic color casts so that objects that appear white in person are rendered white in the image; black-level calibration refers to the brightness of the images making sure they do not seem too dark. This feature-packed camera module includes everything needed to be a robust image capture module for our IoT system.

4.4 Other Considerations

One important aspect of this system is keeping it affordable to the end user or consumer. All the hardware described in this chapter was purchased in one package, available from ArduCAM's website. The price for both the ArduCAM IoTai development board and OV2640 camera was $20. This price for the amount of hardware potential is impressive. While the development board with integrated peripherals was bought for this work to more easily develop primary base functionality of an IoT system, the core pieces of this can be cheaply bought individually or in smaller packages if simpler sensor nodes without imaging capabilities are needed.

Figure 17: ESP32-DevKitC Breakout Board

One example of this is the ESP32-DevKitC, by Espressif, shown in Figure 17. It is a breadboard friendly ESP32 chip with pins broken out, starting at $10. Micro SD card readers and parallel camera interfaces can be purchased quite cheaply, too.

Chapter 5

DEVELOPMENT

5.1 Functionality Overview

This chapter describes the chronological development of significant functions within the IoT system. Each section discusses how a specific system function was created in code and attempts to explain why certain decisions were made in reference to the system goals and requirements. The following topics are presented: the embedded environment used to develop firmware, image capture processes for the sensor node, two viable storage options and respective interfacing, BLE theory, ESP32 BLE functionality for sensor and gateway nodes, low-power sleep modes for the sensor node, and Wi-Fi connection to the Firebase cloud on the ESP32.

5.2 Embedded Development Environment

The firmware developed and tested in this thesis was written in the Arduino Integrated Development Environment (IDE). This environment was chosen as it has the least number of obstacles to begin using and understanding. The Espressif ESP-IDE is also a viable choice for the more experienced embedded developer but has a steeper learning curve than the Arduino IDE. The Arduino IDE is a familiar software environment for anyone who has tinkered with the open-source hardware platforms Arduino provides. The environment is incredibly popular around the globe and has a significant developer community behind it. This means there are many forums, tutorials, and assistance available to get started developing with this IDE.

The Arduino IDE does not come with the ability to interact with the ESP32 SoC or ArduCAM development board automatically. Luckily, there is an easy way to install the board and processor framework to begin writing code on the ESP32. Once the latest version of the Arduino IDE is downloaded and installed on a local machine, a JSON file can be entered into the “Additional Board Manager URLs” field, found at Files->Preferences. The specific JSON file for this specific board can be found here: https://www.arducam.com/downloads/esp32_uno_psram/package_ArduCAM_ESP32_PSRAM_index.json

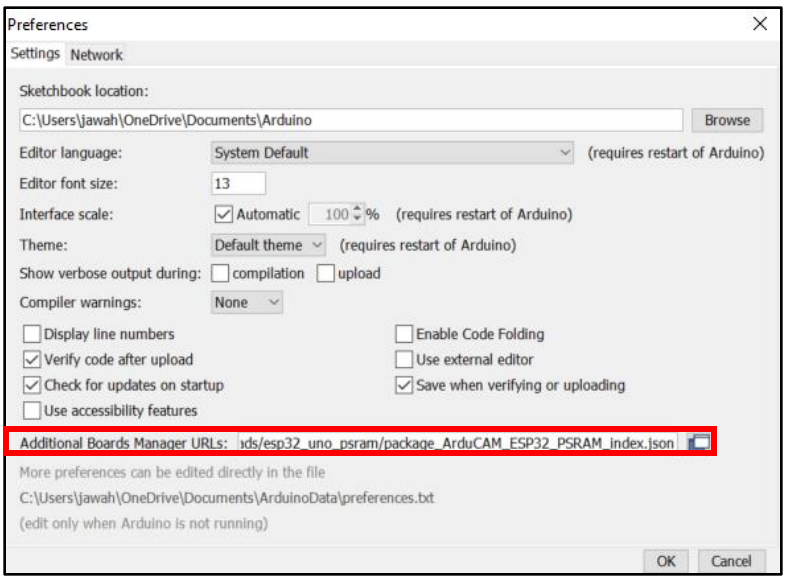

Figure 18: Additional Board Manager URL Field

Enter the file URL into the field shown in Figure 18, separating multiple URL's with commas. This file contains all the necessary links and tools to use all the features on the ArduCAM board. After this is entered, the ESP32 platform package must be installed in the "Boards Manager" menu, found at Tools->Boards->Boards Manager.

Figure 19: ESP32 Platform to Install on Board Manager

The package to install is shown in Figure 19, titled "ArduCAM ESP32S Boards". Once these steps are done and the Arduino IDE is restarted, code development can commence. To develop code for this system, many hours were spent digging into and reading the library files within the ArduCAM GitHub site found at

https://github.com/ArduCAM/ArduCAM_ESP32S_UNO_PSRAM. The site has many C and C++ source and header files for drivers and API's to make interacting with the development board electronics a little easier.

5.3 Image Capture – Sensor Node

The first function to test and implement is to capture an image using the OV2640 camera module, as the entire IoT system revolves around this visual sensing functionality. The goal is to efficiently capture quality images to be used in computer vision processes down-the-line. To develop this, a few files are especially helpful: esp_camera.h, arducam_esp32s_camera.h, sensor.h. The first file provides a camera configuration structure as well as a data structure for the camera frame buffer. The second defines specific API function calls for camera initialization and other key actions. The third provides structure of the OV2640 driver and different settings available to the image sensor.

The basic idea of the function flow is depicted in Figure 20 below. After powering up and booting the processor, we must detect and initialize the camera module to be sure it is ready and capable to capture an image. Once the image is captured, we must provide pointers to the image frame buffer and the image frame length (or size). The buffer will hold the entirety of the image data while the frame length holds the size of the image, in bytes. These two pieces of information will be necessary in storing the imagen in either flash storage or a microSD card. A boolean flag "ESP_OK" is also returned to represent the success or failure of capturing an image.

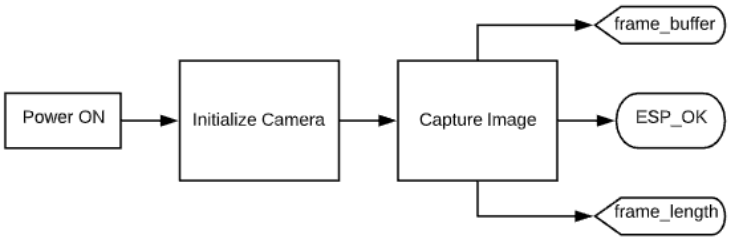

Figure 20: Image Capture Firmware Flow

There are three useful already built API functions that will help. The camera initialization function in Figure 21 is housed within arducam_esp32s_camera.h and configures the camera over the I^2C interface and allocates frame and direct memory access (DMA) buffers. It takes a configuration data structure as input

holding GPIO pin numbers for camera lines, clock frequency and timing channels, pixel format, and frame size information. This function returns a success flag if the initialization operations are completed.

```
esp_err_t arducam_camera_init(pixformat_t PIXFORMAT);

sensor_t * arducam_camera_sensor_get();

camera_fb_t* arducam_camera_fb_get();
```

Figure 21: API Function Names with Parameters

The camera sensor has a control structure defined in sensor.h that can be obtained by using the second function in Figure 21. With this structure, we gain the ability to vary and set different sensor actions while the camera is already initialized. These include brightness, contrast, frame size, gain controls, and many more.

The third function captures an image by obtaining a pointer to the frame buffer that holds the entirety of the image data. The frame buffer data structure contains five variables: a buffer, length, width, height, and format. The combination of these fully describe the image.

The following two custom functions seen in Figure 22 and Figure 23 were developed using the knowledge and API functions described in this section. One is for camera initialization and the other is to capture an actual image. As a note, the print statements are for serial debugging purposes and to make the code easier to read but will not be present in the final distribution of code to save memory and speed up code execution. All serial debugging was done using the Arduino Serial COM port and PuTTY.

```
// Camera Initialization Func
// ==========================================================
void cameraInit() {
  esp_err_t err = arducam_camera_init(PIXFORMAT_JPEG);
  if (err != ESP_OK) {
    Serial.printf("Camera init failed with error 0x%x", err);
    return;
  }
  else
    Serial.printf("Camera init SUCCESS\n");
    sensor_t * s = arducam_camera_sensor_get();
    s->set_framesize(s, FRAMESIZE_UXGA);  // FRAMESIZE_ + QVGA|CIF|VGA|SVGA|XGA|SXGA|UXGA
    s->set_brightness(s, 0);     // -2 to 2
    s->set_contrast(s, 0);       // -2 to 2
    s->set_saturation(s, 0);     // -2 to 2
    s->set_special_effect(s, 0); // 0 to 6
    // (0 - No Effect, 1 - Negative, 2 - Grayscale, 3 - R Tint, 4 - G Tint, 5 - B Tint, 6 - Sepia)
    s->set_whitebal(s, 1);       // 0 = disable , 1 = enable
    s->set_awb_gain(s, 1);       // 0 = disable , 1 = enable
    s->set_wb_mode(s, 0);        // 0 to 4 - if awb_gain enabled
    // (0 - Auto, 1 - Sunny, 2 - Cloudy, 3 - Office, 4 - Home)
    s->set_exposure_ctrl(s, 1);  // 0 = disable , 1 = enable
    s->set_aec2(s, 0);           // 0 = disable , 1 = enable
    s->set_ae_level(s, 0);       // -2 to 2
    s->set_aec_value(s, 300);    // 0 to 1200
    s->set_gain_ctrl(s, 1);      // 0 = disable , 1 = enable
    s->set_agc_gain(s, 0);       // 0 to 30
    s->set_gainceiling(s, (gainceiling_t)0);  // 0 to 6
    s->set_bpc(s, 0);            // 0 = disable , 1 = enable
    s->set_wpc(s, 1);            // 0 = disable , 1 = enable
    s->set_raw_gma(s, 1);        // 0 = disable , 1 = enable
    s->set_lenc(s, 1);           // 0 = disable , 1 = enable
    s->set_hmirror(s, 0);        // 0 = disable , 1 = enable
    s->set_vflip(s, 0);          // 0 = disable , 1 = enable
    s->set_dcw(s, 1);            // 0 = disable , 1 = enable
    s->set_colorbar(s, 0);       // 0 = disable , 1 = enable
}
```

Figure 22: Camera Initialization Custom Function

Within the function shown in Figure 22, we initialize the image sensor with PIXFORMAT_JPEG parameter to indicate that we want compressed output image data in the JPEG format. This function returns an ESP_OK flag if the process is completed successfully. If anything other than ESP_OK is returned, the specific error is printed to the serial port so the developer can view the reason for failure, then exits the function. Upon successful initialization, we obtain the sensor control structure values and use this to set all camera settings. First the frame size is set to UXGA: 1660 by 1200 pixels. This size is the maximum frame the OV2640 can produce (2MP). At this point all other settings can be manually set. All of the settings

shown in the function are done entirely on the OV2640 image processor. The effects and the chosen values are what produced the best images, but the comments describe possible input options. Most image effects are set to be automatic.

```
// Image Capture Func
// ====================================================
static esp_err_t imCapture() {
  camera_fb_t * fb = NULL;
  int64_t time_dur = 0;
  esp_err_t res = ESP_OK;
  size_t fb_len = 0;
  uint8_t *fb_buf;
  Serial.println("\nCamera Cap Started");
  int64_t fr_start = esp_timer_get_time();
  fb = arducam_camera_fb_get();
  int64_t fr_end = esp_timer_get_time();
  time_dur = (uint32_t)((fr_end - fr_start) / 1000));
  if (!fb) {
    Serial.println("Camera Cap FAILED");
    return ESP_FAIL;
  }
  else {
    Serial.println("Camera Cap DONE");
  }
  fb_len = fb->len;
  fb_buf = fb->buf;
  arducam_camera_fb_return(fb);
  Serial.printf("%u Bytes took %ums\r\n to cap", (uint32_t)(fb_len), time_dur);
  return res;
}
```

Figure 23: Image Capture Custom Function

An image capture function was developed around the API functions for our specific purposes, seen in Figure 23. Appropriately sized variables are initialized as null or zero valued for the frame buffer characteristics and timing, seen in the first five lines of the function in the figure above. The variable “time_dur” holds the difference of “fr_end” and “fr_start”. These start and end variables hold the number of microseconds since the ESP32 has booted up both directly before and after the image frame is captured. This allows for the collection of important functionality timing information for power analysis of the system later on. If the function is unable to capture an image, an ESP_FAIL flag is returned. Otherwise, the function will return ESP_OK. Now, we have a camera frame buffer data structure with important attributes we need: the buffer data and the image length. We save these two attributes to our variables “fb_len” and “fb_buf” using the arrow operator.

At this point the image capture function is almost complete. The length and buffer data attributes must be either stored on the device for transmission later on or immediately transmitted off the device. The decision to store the image data on the device for future transmission is made for a few reasons. First, the system being developed must be as scalable as possible. As more imaging nodes are added to the system, a device may not be able to immediately transmit, or the recipient may not be ready to receive data, so organized immediate storage is the best option. This is where either flash or microSD storage can be used. The development of the remaining camera functionality is done without a final decision between these two storage options. The code needed to interact with both storage options is essentially the same with the use of the serial peripheral interface flash file system (SPIFFS) and the SD card file system libraries. The implementation of both is shown in the following section. A more in-depth comparison of the two storage types can be found in Chapter 6.

5.4 SD Card and Flash Storage Interface

5.4.1 SanDisk Industrial 8GB Card

The microSD card chosen for this work is the SanDisk Industrial 8GB card. This is a popular card chosen primarily for its low price and industrial specifications. Commercial microSD cards (regardless of manufacturer) have an operating temperature range of about 0 to 70°C (or 32 to 158°F for us Americans). The limiting factor is the low end of the temperature range as our application of this device is in crop fields where it can easily get below freezing, especially in the winter. The industrial grade cards produced by SanDisk are ideal for this application. The operating temperature range of the card being used is -25 to 85°C [25]. This range includes any temperature we would expect to see crop fields to possibly exist in. Unfortunately, this industrial grade card comes at a higher price tag. An 8GB Industrial card can be bought for around $9 while the commercial counterpart is priced around $5.

5.4.2 Image File Reading and Writing

In order to read and write data as files into storage, the free file system wrapper FS.h developed by Espressif for the ESP32 was used in conjunction with the SD_MMC.h file (for SD) and SPIFFS.h file (for flash). The file system wrapper defines common file functions like mkdir, rmdir, open, close, read, and write. This wrapper is what will help us keep these image files organized and easy to access. The two

header files are drivers for the ESP32 to interact with the on-board microSD card reader and the on-board flash storage.

```
fb_len = fb->len;
fb_buf = fb->buf;
File f = SD_MMC.open(PATH_NAME, 'w');
if (!f) {
  return ESP_FAIL;
}
f.write(fb_buf, fb_len);
f.close();
```

Figure 24: Image Data File Writing Code

The code in Figure 24 shows the basic idea of taking the previously gathered image data and writing it to a file on the microSD card. The PATH_NAME will include the path directory with the wanted file name (for example: /ArduCAM/image_number.jpg). Directories on the card can be made manually with a file explorer on a PC or in code using the SD_MMC.mkdir(). Directories can only be made in flash memory using the SPIFFS.mkdir() call.

Using this functionality, the image data can be stored on the microSD card or in flash. Initially, a single capture was taken and sent to the SD card in order to easily view the image on a PC. The resulting image was surprising. It was dark and had bad white balance showing a green tint. Originally, the thought was that the automatic settings would do a better job at correcting the image. After some testing, it was found that the images would look best if they were taken in series of around five at a time, where the quality of the image increased until image three. After the third image, the pictures would look balanced and well corrected. This increase in quality of images from the first to the last image in one series of five is shown in the Figure 25.

Figure 25: First through Fifth Image Taken in Series with OV2640

These five images were taken with no change in code or settings, just within a series of images taken on the same board boot and camera initialization. This is important because we need to be able to

consistently save and transmit the best image available. Therefore, it is best to take image captures in series of at least five and save the last one to storage for transmission.

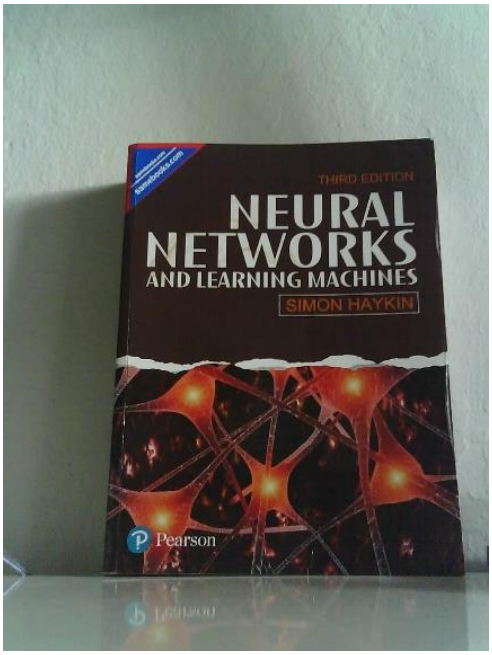

Figure 26: Image Captured by OV2640 (rotated 90 degrees), 139KB

Figure 26 shows an image captured with a textbook in view. This shows the color saturation, brightness, and balance are all sufficient using the automatic settings. It is an impressive image for a 2-megapixel simple camera module. The 1600 by 1200 pixel image file is compressed to 139KB by the JPEG engine, a compression ratio of almost 14.

Theoretically, the 8GB microSD card could hold over 57,000 equivalent images. While this seems like an impractical metric for the point-to-point system being developed for this work, the system needs to be scalable. This large storage space would allow for many sensor nodes to save and share image data between themselves frequently before finally routing the image to the gateway. This also allows space for other types of data along with images to be saved. Prediction systems in the future will likely need more than just image data to be robust. The ability to save other data types like temperature, soil moisture, and airflow data could prove useful.

The 4MB chip of flash storage on the development board would be able to hold around 28 equivalent images. While the flash storage provides much less space to store image data, it could still be effectively used in this system. The idea for the sensor nodes is to not keep an image file for very long. Once the image

has been transferred off the device, the file can be removed as it is no longer needed. By constantly transmitting and subsequently removing image files, flash storage is still a viable storage option.

Though the microSD storage seems much better for our system at this point due to its easily expandable memory, a disadvantage is mentioned in the following section. The use of flash storage also has a major advantage compared to a microSD card in terms of power, discussed in chapter 6.

5.4.3 Challenges

Using the microSD card for storing images on this development board poses an interesting problem that came up while monitoring system current draw. Surprisingly, microSD cards are actually very power-hungry devices. They require large currents to read and write data, up to 100mA for the SanDisk being used [23]. These read and write times are relatively short compared to the amount of time the card will sit idle and unused. Unfortunately, the idle current draw for this type of storage device is around 500µA, constantly pulled when unused. While this current value sounds small, the sensor node will spend the vast majority of its operational lifetime trying to use as little current as possible to extend battery life. The development board has a dedicated voltage regulator for the microSD card reader, meaning the card will continuously be powered even when the ESP32 is in a low-power mode. This problem is addressed in more detail in Chapter 6.

5.5 Bluetooth Low Energy (BLE) Functionality

5.5.1 BLE Theory and Overview

In order to develop firmware that follows the Bluetooth Low Energy standard, one must understand basic theory and operation of the wireless communication. BLE is often the ideal choice for devices that want to send small amounts of sensor data while using minimal power over short distances. Our use-case of BLE is slightly unusual, as we want to transmit relatively large image files in infrequent bursts. The primary reason we are using BLE rather than classic Bluetooth is because of the flexibility in network topologies. BLE supports point-to-point, broadcast, and mesh topologies while the classic only supports the first. BLE is chosen to make this system as scalable and adaptable as possible for the future.

With BLE, there are two types of devices: the server and the client. The ESP32 SoC can act as either of these roles. In general terms, server is the device that is holding data it wants to share while the client is the

device that wants to receive this data. Figure 27 below depicts the two types of devices within BLE and the connections between them.

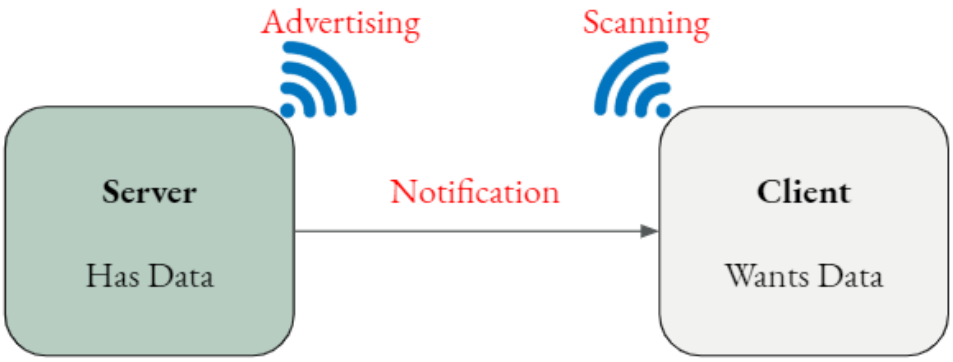

Figure 27: BLE Basic Server and Client Interaction

When the server has control of the data it wants to share over the air, it can "advertise" its existence to clients that happen to be in the general vicinity. A client can "scan" the area and attempt to find a connection to a server in order to receive data. Once a verified connection is made, the server can proceed in one of two directions. One is to send data without letting the client know, forcing the client to poll for data. The other is for the server to immediately update the client when it sends data using either "notifications" or "indications," both of which are shown in Figure 28.

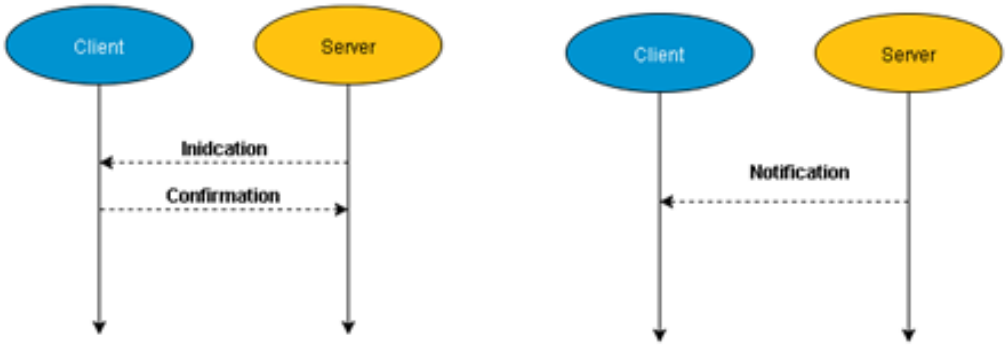

Figure 28: Indication / Notification Schemes

Notifications from the server transmit data continuously with no expectation of a reply from the client. This means the server can send data to the client without the client's acknowledgment. An indication on the other hand, must wait for an acknowledgement from the client that it received the transmitted data before it can send the next data packet. The notification scheme is faster (but possibly more error prone) than the indication scheme. This describes the behavior of a point-to-point BLE topology. In a mesh network, all devices would be connected, making it a many-to-many connection.

Now, how does each device know which other devices it should connect to? And what is the data format for packets that are being transmitted? Both of these questions can be answered by studying the Generic Attribute (GATT) hierarchical data structure that sits on top of the BLE layers, depicted in Figure 29. This structure defines basic models and procedures to allow devices to discover, read, write, and push data elements between them [24]. This is important to understand as it will help us use BLE and write our firmware.

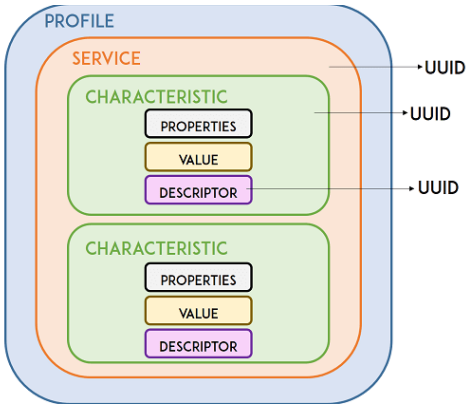

Figure 29: GATT Data Structure for BLE [25]

The topmost layer of this data structure is the Profile, which technically does not exist on the actual BLE peripheral itself, the ESP32 in this case. The Profile is a pre-defined collection of Services that have been compiled by either the Bluetooth SIG or by other designers.

BLE Services are used to break up data being held into logic entities. They hold specific chunks of data called Characteristics (Services can contain more than one). Each Service is able to distinguish itself from others by using a unique numeric identification number, called a UUID. These identification numbers can either be officially adopted 16-bit values or completely custom 128-bit values. An example of this is the officially adopted Heart Rate Service (UUID: 0x180D) which contains three Characteristics: Heart Rate Measurement, Body Sensor Location, and Heart Rate Control Point. These are commonly used in medical or fitness wearable devices, like Fitbit watches.

The lowest level concept of a GATT structure is the Characteristic. The Characteristics encapsulates a single data point or array of data from the server device in its “value” field. Like Services, Characteristics identify themselves through the use of UUID’s, also either 16-bit (officially adopted) or 128-bit (custom). The custom UUID’s are used in this thesis to ensure that only the devices within our network can

understand this data. The Characteristic value is the main field that will interact with the BLE client and server; it is able to be written to and read from. The value has no restrictions on the type of data it can contain but is limited to 23 bytes of payload data, defined by the GATT structure maximum transmission unit (MTU) [26].

While this background does not make up all the theory or information that goes into BLE procedures and protocols, it is enough to understand the development procedure used in this thesis. New terms and concepts that are used and not mentioned above will be explained as they are brought up.

5.5.2 Image Transmission via BLE – Sensor Node

The development of a custom BLE image transmission process for the ESP32 involved reading and understanding many source and header C files within the BLE library in GitHub, in addition to the theory described in the previous section. The library source files can be located at: https://github.com/espressif/arduino-esp32/tree/master/libraries/BLE/src. The Espressif API online manual also provides documentation of functions used within the library files for the Arduino ESP32 core.

5.5.2.1 General Overview

The firmware flow diagram in the Figure 30 depicts the high-level process of BLE image data transmission from the time of boot up and assumes there exists a specific image file we want to transmit. After capturing an image and saving it to storage, the process starts by defining the custom Service and Characteristic UUID values. Then the server is initialized as a BLE device, creates the Service and Characteristic properties, then begins to advertise itself over-the-air. It continues advertising until a stable connection is made to the correct client. Once this occurs, the storage file system is checked to see if the file we want to transmit exists. If it does, we open the file for reading. While we can still read new data from the file, we enter a custom algorithm that fills a buffer of any size (MTU less than or equal to 23 bytes) for any size image captured. Each time the buffer is completely filled, the Characteristic value is set before a BLE notification is sent, until the end of the file is reached. The process ends with the de-initialization of the file system and the BLE hardware. By doing this as soon as the transfer is complete, we can save as much valuable power as possible.

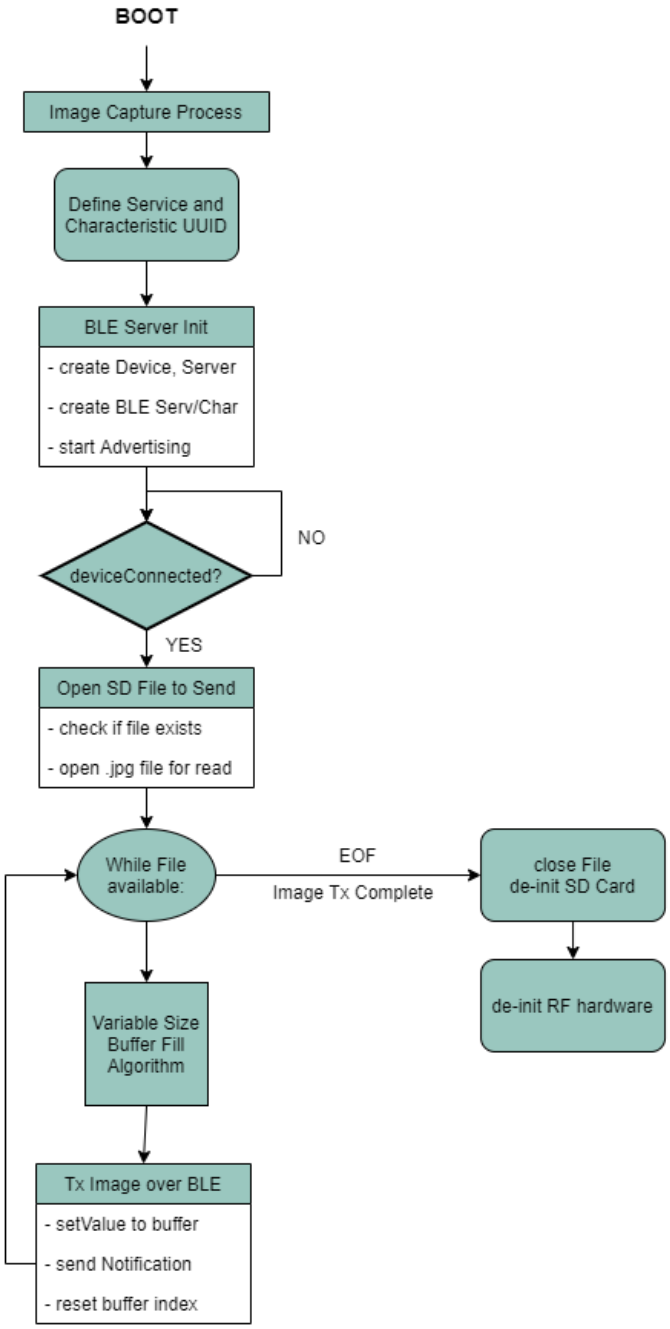

Figure 30: BLE Data Transmission Firmware Flow Diagram

5.5.2.2 BLE Device Creation and Advertising on ESP32

To define the UUID's for both the Service and Characteristic, we use a custom UUID generator from https://www.uuidgenerator.net. The values generated by this site for the Service and Characteristic UUID values respectively are:

4fafc201-1fb5-459e-8fcc-c5c9c331914b AND beb5483e-36e1-4688-b7f5-ea07361b26a8

Next, the actual BLE device instance is created with a name of our choice; “ESP32” is chosen for simplicity. This is the name that will be displayed for other devices to view it. The BLE server, Service, and Characteristic with all properties are then created. Figure 31 shows the code to instantiate these. The properties chosen to be included with the Characteristic are READ, WRITE, NOTIFY, and INDICATE. These are all properties that can be enabled but do not necessarily all have to be used for a specific case.

```
// Create the BLE Device
BLEDevice::init("ESP32");

// Create the BLE Server
pServer = BLEDevice::createServer();

// Create the BLE Service
BLEService *pService = pServer->createService(SERVICE_UUID);

// Create a BLE Characteristic
pCharacteristic = pService->createCharacteristic(
                  CHARACTERISTIC_UUID,
                  BLECharacteristic::PROPERTY_READ   |
                  BLECharacteristic::PROPERTY_WRITE  |
                  BLECharacteristic::PROPERTY_NOTIFY |
                  BLECharacteristic::PROPERTY_INDICATE
                  );
```

Figure 31: Creation of BLE Device, Server, Service, and Characteristic

To start advertising our newly created BLE device and its properties, we use the BLEDevice and BLEAdvertising header and source code files. We can use the function call getAdvertising(), a device file method, to retrieve an advertising object. Advertising has many different settings we can manually change, but for now we can get by with using the basics. First, we add the service UUID to the advertising object to be sure other devices are able to recognize it. Then we can start the act of wirelessly advertising our device for BLE clients to see and attempt to connect to. The code to start advertising is shown in Figure 32. It is important to note that once advertising starts, the power consumption of the ESP32 will drastically increase. This action must only be initiated if the device is completely ready to communicate with a potential client.

```
// Start advertising
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(false);
BLEDevice::startAdvertising();
```

Figure 32: Start BLE Advertising

Once advertising has started, we must determine when our specific client has found our server and when it has made a complete connection. In order to do this, we use a callback function. After the creation of the Server, we can add in a line of code to set its callback as a custom function we name MyServerCallbacks(). This callback function is only triggered or entered when the Server connects to a client successfully. So, when the Server and Client make that stable connection (or disconnection), we are able to write any code to run as soon as this happens. To separate the connection and disconnection alert actions, we define two different custom functions within the MyServerCallbacks, OnDisconnect() and OnConnect(). Inside the connection function, we set a Boolean flag called deviceConnected to true. This flag enables code to run that will format and transmit chunks of image data.

5.5.2.3 BLE Data Format and Transmission on ESP32

The code developed to format and transmit the data sits within an adaptive buffer filling process. Figure 33 shows a detailed flow of the firmware developed to format and transmit image data. As a reminder, the chunkSize variable can be changed to any size from 1 to 23 bytes. This limitation comes from the GATT maximum transmission unit (MTU) in the BLE standard protocol, mentioned in the BLE theory section. Because it may be useful to have a flexible MTU size, the code is written in a way to adapt to this as the image file size will most likely not be perfectly divisible by the chosen buffer size. The size of the image to transmit will also never be the same size.

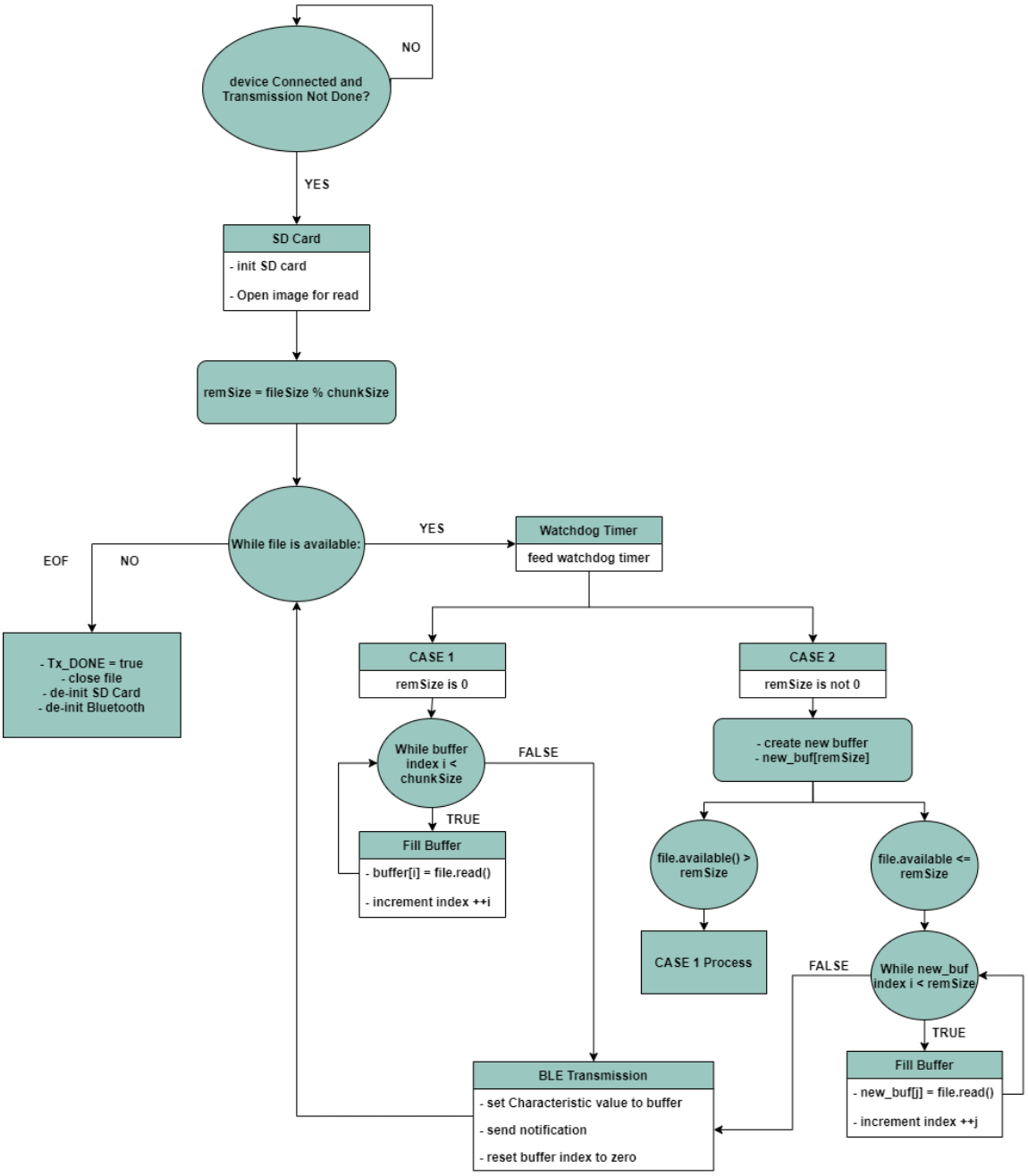

Figure 33: Data Formatting and Transmission Firmware Flow

First, the device must decide it is ready to transmit to a client by checking the connection status. If the deviceConnected flag is true and the Tx_DONE flag is false, we enter our first statement. Once inside this conditional, we initialize the storage file system and check to see if a file exists that is ready to be transmitted, and open that file for reading. Following this, a calculation is done to determine whether a new buffer should be created for the last data packet transmission. The size of this buffer is equal to the

remainder of the file size divided by the MTU, chunkSize. This operation is done using the modulo operator. If the result is not zero, that means the final buffer transmission will be unequal to the MTU size. This result will vary between every image capture and is stored in a variable called remSize. Next, we want to continually read the image data and format buffers until the end of the file. To do this, we use a file system method called "available." This method checks to see how many bytes are available for reading from the file and returns zero when it is at the last byte of data in the file. Depending on the result of remSize, we enter one of two case statements to continuously fill buffers, write Characteristic values, and send notifications to the client device so it is aware of new data. To set BLE data values and send a notification, Figure 34 shows the two lines of code that are used.

```
pCharacteristic->setValue(buf, sizeof(buf));
pCharacteristic->notify();
```

Figure 34: BLE Characteristic Value Setting and Notification Code

The only time the new buffer size determined by remSize is utilized is just before the last notification is sent for the last data packet transmission. Once the image file has been transmitted, we set the Tx_DONE flag, close the file, de-initialize the storage system, and de-initialize the Bluetooth controller on the ESP32. This stops all processes required to operate these functions on the CPU and most importantly stops the power-hungry BLE RF hardware.

This entire described process is best illustrated through an example. Figure 35 and Figure 36 display the serial port while the device is running, and the printed statements show what is taking place. During this example, the chunkSize was set to 4 bytes of image data.

```
13:48:08.324 -> image_1.jpg: 118153 Bytes, 620ms to capture and write
13:48:08.324 -> Picture 1 Captured
13:48:08.629 -> image_2.jpg: 116695 Bytes, 301ms to capture and write
13:48:08.629 -> Picture 2 Captured
13:48:08.900 -> image_3.jpg: 109692 Bytes, 290ms to capture and write
13:48:08.900 -> Picture 3 Captured
13:48:09.072 -> image_4.jpg: 62318 Bytes, 164ms to capture and write
13:48:09.072 -> Picture 4 Captured
13:48:09.245 -> image_5.jpg: 61600 Bytes, 157ms to capture and write
13:48:09.245 -> Picture 5 Captured
13:48:09.960 -> Advertising Started
13:48:20.523 -> Connected to Client
13:48:20.592 -> File size: 61600 Bytes, MTU size: 4 Bytes, remSize: 0 Byte(s)
13:48:21.071 -> CASE 1 Entered
```

Figure 35: Case 1 Process in Serial Port

In Figure 35, a series of five images are captured and the advertising period is started. Once a connection is made to the client, the sizes are printed of the file to be transmitted, the current MTU size, and the remSize. Because the file size (61600 Bytes) was perfectly divisible by the MTU (4 Bytes), case 1 is entered, and the image file is transmitted using a single size buffer.

```
13:51:47.758 -> image_1.jpg: 121158 Bytes, 631ms to capture and write
13:51:47.758 -> Picture 1 Captured
13:51:48.061 -> image_2.jpg: 119193 Bytes, 304ms to capture and write
13:51:48.061 -> Picture 2 Captured
13:51:48.365 -> image_3.jpg: 108839 Bytes, 289ms to capture and write
13:51:48.365 -> Picture 3 Captured
13:51:48.536 -> image_4.jpg: 62883 Bytes, 165ms to capture and write
13:51:48.536 -> Picture 4 Captured
13:51:48.808 -> image_5.jpg: 61017 Bytes, 250ms to capture and write
13:51:48.808 -> Picture 5 Captured
13:51:49.487 -> Advertising Started
13:51:54.768 -> Connected to Client
13:51:54.834 -> File size: 61017 Bytes, MTU size: 4 Bytes, remSize: 1 Byte(s)
13:51:55.313 -> CASE 2 Entered
```

Figure 36: Case 2 Process in Serial Port

In Figure 36, a different series of five images are captured. This time, however, the file size (61017 Bytes) is not divisible by the MTU (4 Bytes). There is one byte of data remaining, so the buffer structure must be of a different size and case 2 is entered. This process adjusts to any appropriately size MTU and any size image file to ensure full file transmission.

By changing the value of the MTU (chunkSize), we can enable higher data throughput for our system. When this code was initially tested with using a minimum chunkSize of one-byte, full image transmission typically took around one minute. This time duration must be minimized as wireless transmission has the highest power consumption out of all the CPU functions, so the MTU should be near or at the limit of 23 bytes. Chapter 6 discusses the optimization and analysis of this image data throughput.

5.5.3 Image Reception via BLE – Gateway Node

5.5.3.1 General Overview

Code must be developed to create a BLE client on the gateway node to receive this image data being transmitted. This functionality was first done using an app called nRF to confirm operation of the BLE server. Then, a BLE client is configured on the ESP32 gateway node. This section describes both processes.

5.5.3.2 BLE Client Testing with nRF Application

Initially, to test the functionality of the BLE server, the Nordic nRF android application was used on a smartphone. Instead of developing a client device on another ArduCAM development board and having to debug both sides at the same time, one can use this application to have their smartphone act as a client to view necessary BLE information. This method reduces the amount of debugging needed and provides a quick and efficient way of testing the server-side functionality. Once the nRF is installed and opened, navigate to the “Devices” section by tapping the 3 bars in the upper left-hand corner, shown on the left of Figure 37.

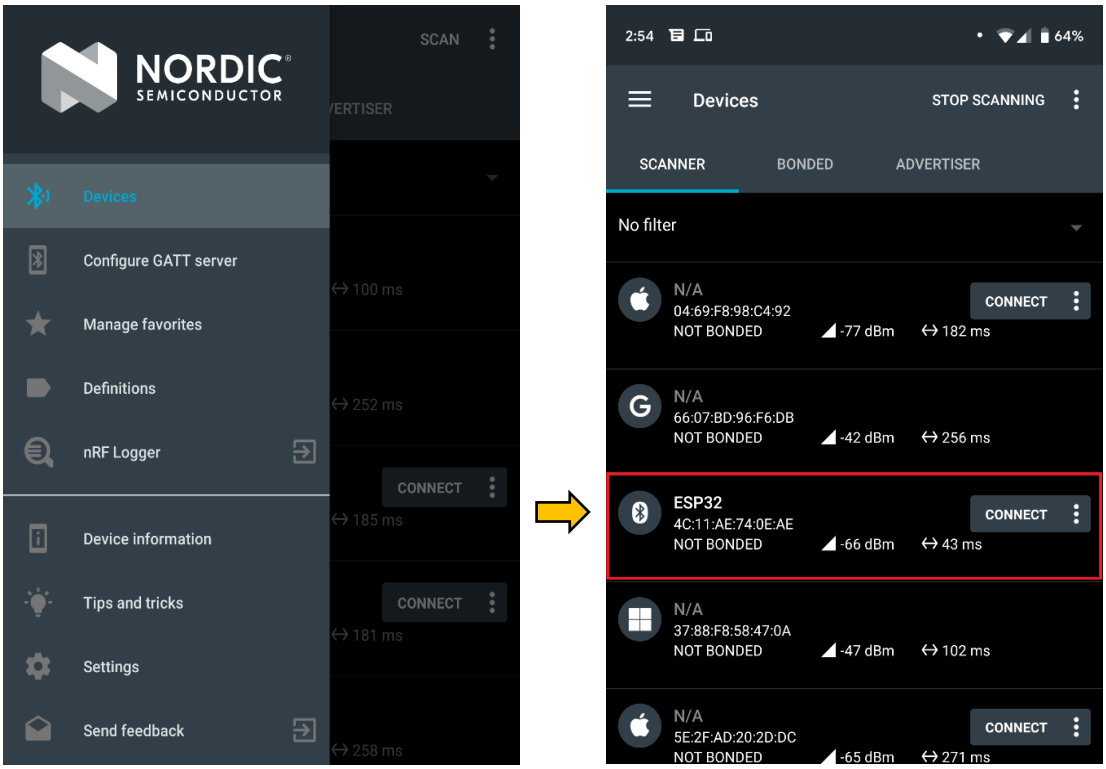

Figure 37: Nordic nRF BLE Connection Screenshots

This will take the user to the scanning page, where all BLE devices currently advertising within range will be listed, shown on the right of Figure 37. Each device will have a name (we called ours “ESP32” in our code) and if applicable, an option to connect to it. By connecting to our ESP32 device, we trigger our board to transmit image data, as written in our server code. nRF allows us to see the Service and Characteristic UUID values and the current Characteristic value in real-time.

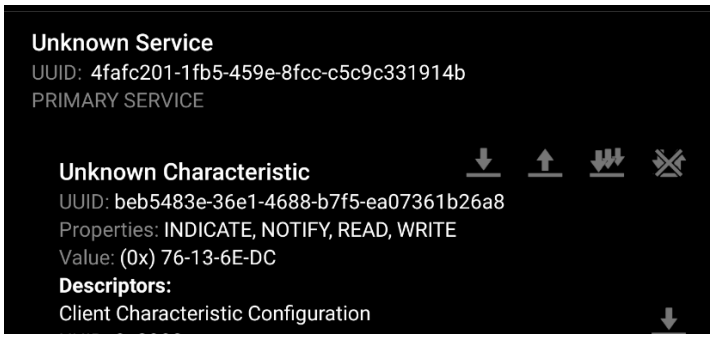

Figure 38: ESP32 BLE Device Information on nRF App

As shown in Figure 38, our device is “Unknown” because we have defined our device with custom UUID values. The values in both the Characteristic and Service UUID fields both match the values we configured in code. The Characteristic value shown (0x76-0x13-0x6E-0xDC) is a four-byte chunk of image data being transmitted at that instant in time.

This application allowed for easy debugging of BLE code being developed. The app was used incrementally during all steps of the Sensor Node development to ensure proper operation of code. nRF also gives a few other pieces of information that is useful to a developer, like the transmit power level and advertising interval. Using this method, the operation of the server side BLE firmware is confirmed.

5.5.3.3 BLE Client on ESP32

Now the nRF app must be replaced with an actual client device. By using another ArduCAM ESP32 development board, we can create a client device to receive the server’s data. To create a client device, we only need to reference the BLEDevice.h header file from GitHub. The high level flow of creating a BLE client device and wirelessly retrieving data on the ESP32 is described in Figure 39 below.

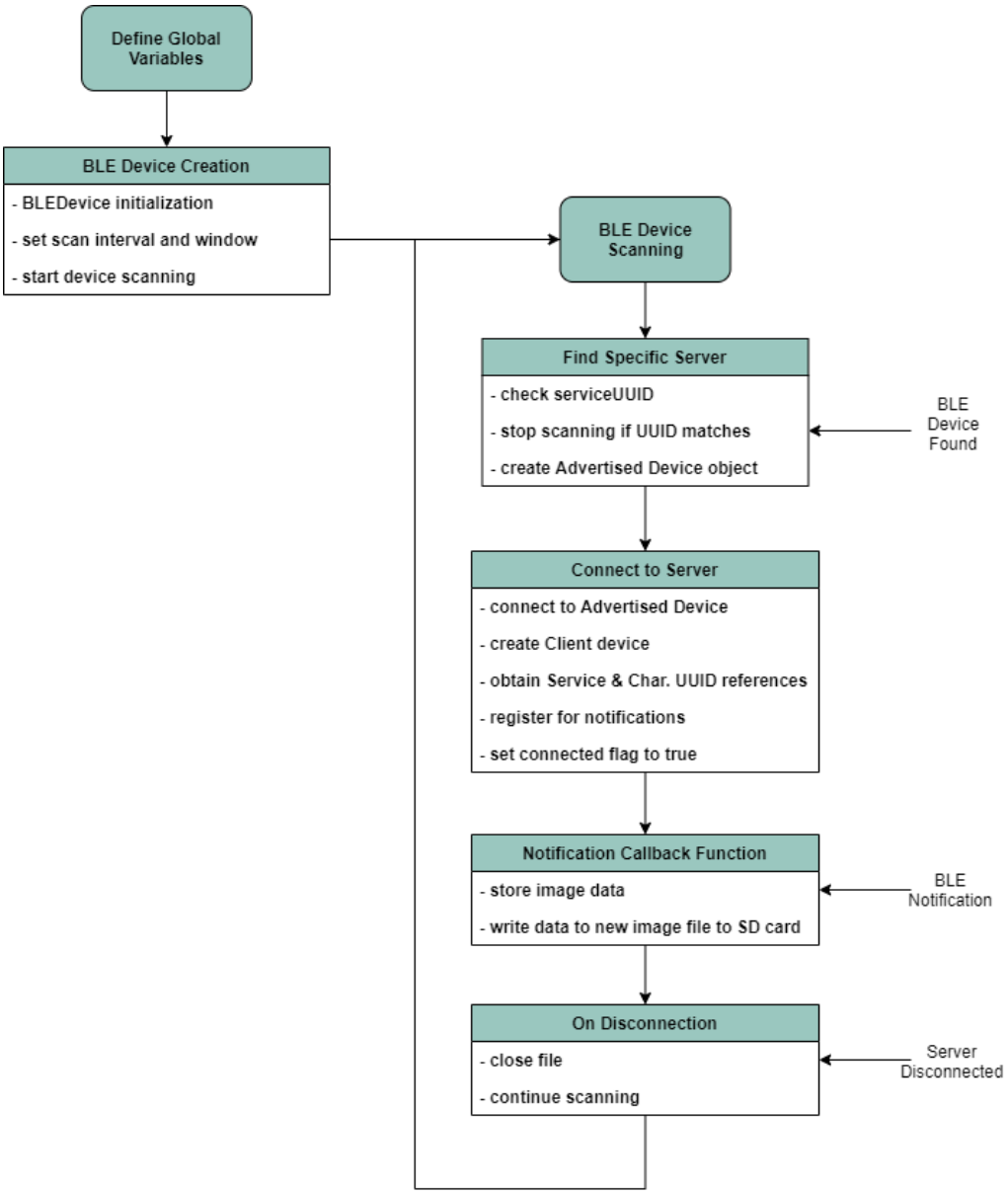

Figure 39: BLE Data Reception Firmware Flow Diagram

First, we define variables that will be needed throughout the client creation process.

- serviceUUID and charUUID: These will hold the Service and Characteristic UUID values of the server we wish to connect to
- Data: This will hold the Characteristic value image data as an unsigned integer
- doConnect: This Boolean flag will determine if we should connect to a found BLE device
- connected: This Boolean flag states if the client is connected to the server
- doScan: This Boolean flag determines if the client should scan for devices

In the setup code, a BLE device is initialized in the same way as with the server. This time, the device does not need to be named as there is no need to connect with it, so the argument is left empty. From this device, we retrieve a scanner object and set the callback function name we want to inform us when we have detected a new BLE device by using the code shown Figure 40.

```
BLEDevice::init("");
BLEScan* pBLEScan = BLEDevice::getScan();
pBLEScan -> setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
```

Figure 40: BLE Device Scan Object Creation

In order to specify scanning characteristics for the client device like the interval and window times, we have to understand what these terms mean. The scan interval is defined as the time interval from when the Bluetooth controller started its last low-energy scan until it begins the subsequent scan. The scan window is the duration of the low-energy scan, meaning it must be less than or equal to the scan interval. After every scan interval, the device changes frequency to the next advertisement channel. An illustration of this is shown in Figure 41. It scans on a different advertising channel after each interval.

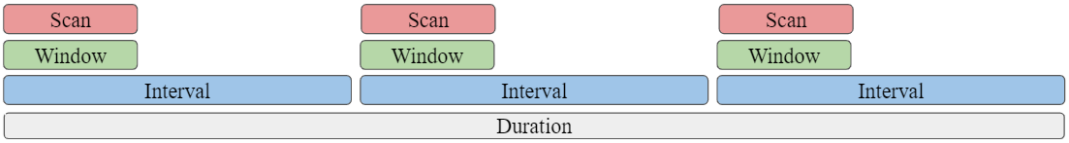

Figure 41: BLE Scan Interval and Window Diagram

Both of these values can be manually set in code. The functions take in integer values with millisecond units. These values were varied and tested but for our application do not need to be at specific intervals or times to find our server. The device actively transmits during the scan window, so the lower the ratio between window and interval, the lower the power consumption by the device. Next, the decision to use either active or passive scanning is debated.

Active scanners respond to every device they hear from, asking if they have more data to send. The devices then respond with any "scan response" data they have been configured to send. When passively scanning, a device will only listen to Bluetooth devices, quietly collecting data about its surroundings. Our server has been configured to not send any scan response, as seen in Figure 32. We will use passive scanning as we can receive all necessary data and use less power than active scans [27].

```
pBLEScan->setInterval(1500);
pBLEScan->setWindow(500);
pBLEScan->setActiveScan(false);
pBLEScan->start(5, false);
```

Figure 42: Set Scan Characteristics Code

With these three settings configured as set in Figure 42, the client can begin to scan for BLE devices nearby. As there are only three channels that BLE devices can advertise at, connection is typically very fast. Our scan duration is set to five seconds, shown in the figure above.

After scanning is started, every BLE device that the scanner finds triggers the MyAdvertisedDeviceCallbacks() function. Within this function, the device checks to see if the acquired device contains the Service UUID that we are searching for. If the device is a match, the BLE device stops scanning and creates a new advertised device object with the found device information.

After this, the code is funneled into a custom connectToServer() function, seen in Figure 43, where an actual client device is created with a unique callback function called MyClientCallback(). This callback can be entered on connection or disconnection from the server. The client connects to the advertised device object and obtains references to the Service and Characteristic UUID's. Lastly the device registers for notifications (rather than indications) by defining a notifyCallback() function to trigger when the server has new data within its Characteristic value. The connection flag is then set to true.

```
bool connectToServer() {
  BLEClient*  pClient  = BLEDevice::createClient();
  pClient->setClientCallbacks(new MyClientCallback());
  pClient->connect(myDevice);
  BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
  if (pRemoteService == nullptr) {
    pClient->disconnect();
    return false;
  }
  pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);
  if (pRemoteCharacteristic == nullptr) {
    pClient->disconnect();
    return false;
  }
  if (pRemoteCharacteristic->canNotify())
    pRemoteCharacteristic->registerForNotify(notifyCallback);
  connected = true;
}
```

Figure 43: Connect to Server Function Code

Every time the server sends out a notification of new data being written to its Characteristic value, the client enters the notifyCallback() function, as Figure 44 depicts. Within this code, we use the reference to the Characteristic to obtain its value and size. Once the data and its length are stored on the client side, the data is written to a new file in the storage file system being used.

```
static void notifyCallback(
  BLERemoteCharacteristic* pBLERemoteCharacteristic,
  uint8_t *pData, // image data
  size_t length,  // length of *pData
  bool isNotify) {
    Data = *pData;
    feedTheDog();
    f.write(pData, length);
}
```

Figure 44: Notification Callback Function

The client side of the system is able to write to the file within the callback function without worrying about the server sending another notification because of the time it takes to format the buffer data on the server side. The time required to read the image data and fill a buffer of any size (greater than one byte)

will always be larger than the time to only write that data to storage. Finally, once the server and client disconnect, the received image file is closed, and the client can continue scanning for other BLE devices.

5.5.4 Server to Client BLE Link Verification

After the code was developed for receiving data on the client side of the system, verification was done to confirm the operation of the entire server-to-client BLE link. This was done both qualitatively and quantitatively. First, the image file written to the client's microSD card was opened on a PC to be sure that all the JPEG formatting data was intact and was able to be opened and viewed by an end user. Next, the server and client image files were opened side-by-side in an online Hex editor to compare raw JPEG image data. The notable areas of concern when comparing raw data files were the first and last two bytes. Every single JPEG image starts with 0xFFD8 and ends with 0xFFD9 to signify the beginning and end of the file. These two points helped determine if the entire image was being transmitted.

5.6 Low Power Sleep Modes

The most important part of the sensor node's operation is how it manages itself while not capturing, processing, and transmitting data. The sensor device will spend the majority of its time doing nothing, then have short bursts of large power consumption to capture and store images then transmit them over Bluetooth Low Energy. For this application, the time between bursts of activity is the frequency at which images are captured. Whether that is once an hour or once a week, this down time in-between will be a significant percentage of each period. The diagram in Figure 45 below describes this timing.

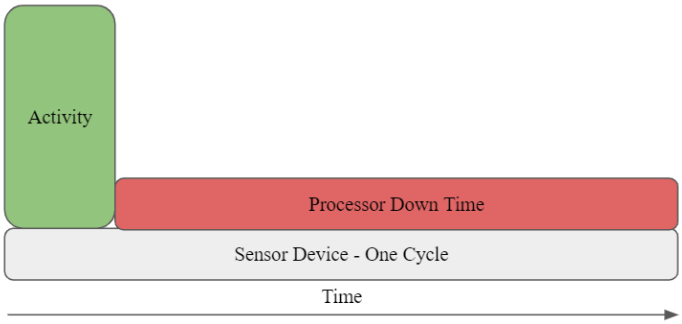

Figure 45: Sensor Device Cycle Timing Diagram

The amount of time that the processor is doing nothing for this application will be quite high as our frequency of image acquisition will be quite low. The computer vision and machine learning pipelines will

not need images taken every minute of every day. The frequency will be more along the lines of a few times per day. This means the duty cycle of processor "on" time will be very low, much less than 1% of the time. In order to save as much power during the other 99% of down time, we take advantage of processor sleep modes.

The term "sleep mode" refers to a condition in which an embedded system enters a low-power hardware mode. Typically, the microcontroller ceases to perform any computations and shuts down any peripheral functions. While in sleep mode, the processor may still retain some stored data and will power specific timers and interrupts so it can wake up again.

In our system, the ESP32 SoC offers a variety of configurable power modes, each of which has its own distinct set of power saving features and capabilities. Under normal operation, the chip runs in "Active Mode." This mode keeps everything powered at all times (unless specifically turned off in code). This includes the Wi-Fi module, the processing cores, and the Bluetooth module. In this application, it is wise to save as much power as possible during the down time, so we look to enter the "Deep Sleep" mode.

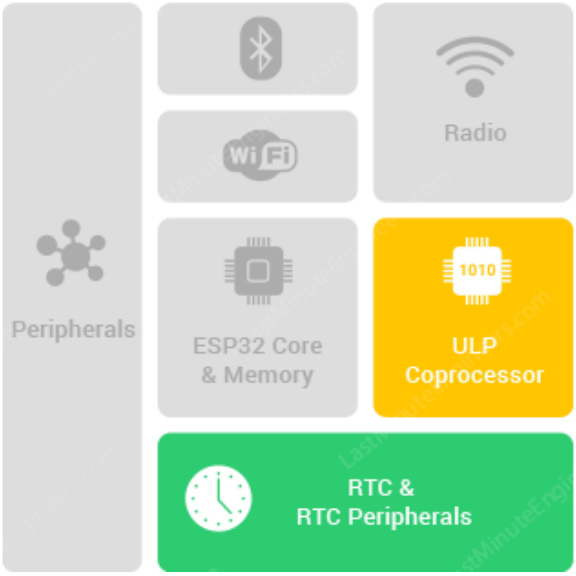

Figure 46: Deep Sleep ESP32 Powered Components [28]

In deep sleep mode, the CPU, most of the RAM, and all of the digital peripherals are powered off, depicted as gray boxes in Figure 46. The only parts of the chip that remain powered on are the real-time clock (RTC) controller, the RTC peripherals, the ultra-low-power (ULP) co-processor, and the RTC memory, depicted as colored boxes in Figure 46. During this mode, the ULP processor is capable of taking

simple measurements and can wake up the main system based on the measured data. Along with the CPU, the main memory of the chip is disabled so everything that is stored in memory is wiped and cannot be accessed. However, RTC memory is kept on, so any variable or data that needs to be preserved between boots can be stored here.

As this system will not need any processing to be done while in sleep mode, the ULP processor can be powered off while in deep sleep. The ESP32 datasheet claims that the chip will run at around 10µA in this configuration. It must be kept in mind that this value is ideal for just the ESP32 SoC and does not account for the other electronics on the ArduCAM development board. Many other developers were unable to realize this current, even with just the SoC. The observed current consumption value of this sensor device will be higher than that of this claim during deep sleep in practice.

To enter deep sleep mode in code, a few things must be done. First, a sleep mode time duration is defined in microsecond resolution in an unsigned 64-bit integer. In theory, this value's maximum quantity would be able to enter the ESP32 into sleep for over 500,000 years, giving us extreme flexibility in choosing a sleep time duration. For our purposes, a conversion factor is multiplied by our time variable, so it represents a number of seconds to sleep instead of microseconds.

```
#define US_TO_S_FACTOR 1000000
#define TIME_TO_SLEEP 86400
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * US_TO_S_FACTOR);
esp_deep_sleep_start();
```

Figure 47: Deep Sleep Functions

The code snippet in Figure 47 above shows the conversion factor and time to sleep variables. In this figure, the time to sleep is set to 86,400 seconds and would sleep for an entire day before waking up. The first function enables this timer used to wake the chip from sleep and should be called in the setup code. The last function will immediately enter the device into deep sleep and wake up in the predetermined amount of time with a full reboot of the device. The sensor node enters deep sleep in the server callback function that triggers when the server and client disconnect from each other.

5.7 Wi-Fi and Cloud Server Link

As the final action in the system, the gateway node will upload aggregated image files to a cloud service. This service will act as an accessible storage space for any end user. The cloud server can be accessed by anyone with an internet connection, on any device, that has the account information.

5.7.1 Firebase Realtime Database

The service chosen to act as the cloud server for this system is Firebase, by Google. While a few different services were researched, including Thingspeak, Microsoft Azure, and Amazon Web Services, Firebase was deemed to be the most favorable. Firebase has a free subscription of 1GB of storage and 10GB of downloads per month. This is more than enough for our use, especially if images being stored are downloaded to a local machine and deleted from Firebase once they are used.

Firebase is an open-source cloud server and because of that, many developers have created custom libraries for popular IoT hardware to interact directly with the storage. A very well-documented Firebase Realtime Database Arduino Library for ESP32 is available for installation on GitHub. It supports many common database functions and most importantly, enables direct file pushing from a device's SD or flash storage.

5.7.2 Firebase Database for the ESP32

In order to start uploading files and data to the Firebase database using the ESP32, a few things must be set-up. First, a Firebase account (can be linked to your Google account) must be made. Next, a database can be created with a custom host name. Once this is made, the Firebase automatically generates an authorization key to limit database usage to only secure users. This key is called the database "secret." Next, the Firebase ESP32 library from GitHub can be downloaded as a zip file and directly included into the Arduino IDE. Once this is done, we can define names and passwords for Firebase and our local Wi-Fi connection in code.

```
#define FIREBASE_HOST "wahl-thesis.firebaseio.com"
#define FIREBASE_AUTH "EiR5zCpZo1pJltn9Y8SkaVUiPNLzM9q9YMIoDOeJ"
#define WIFI_SSID "JDW_Pixel"
#define WIFI_PASSWORD "yszd691#"
```

Figure 48: Firebase and Wi-Fi Access Keys

The database name, “secret”, and local Wi-Fi credentials are shown in Figure 48. This Firebase database is called “wahl-thesis” and I use my phone’s Wi-Fi hotspot credentials to connect. To start uploading image file data via a local Wi-Fi signal, the following process must be completed.

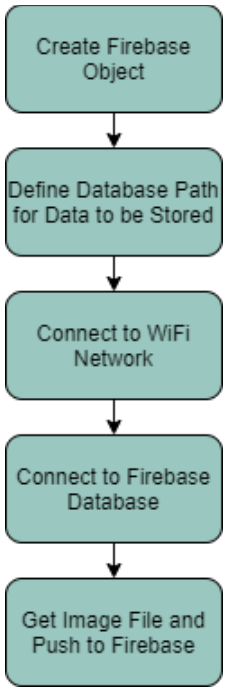

Figure 49: File Upload to Firebase Process

The process described in Figure 49 was followed to test this functionality on a single image stored in flash memory on the gateway node. The database path defined for the data to be stored at is under a folder called “Gateway-Node”. Firebase stores file data on its database by encoding the data stream into base-64. This is a prevalent scheme on the World Wide Web, where it can be used to embed image files or other binary assets inside textual assets (like HTML). Encoding binary data this way makes it more reliable to be stored in databases. This encoded data string stored can then be entered into any web browser to be viewed and downloaded locally, if needed.

Figure 50: Firebase Database Image File View

In the Figure 50, the result of the test on the database side is shown. The file is uploaded as a string of base-64 text. This string can be copied and entered into a web browser, replacing “file,” with “data:image/jpeg;”, and the image can be viewed and subsequently downloaded. While this process is simple enough to be done manually for every image, it would be even easier to automate it. A python script is developed to automatically scrape the database contents, decode the image file strings, download them to a local folder, then delete the file data from Firebase. This script can be found in Appendix B.

To combine this code into the gateway node, the cloud uploading process will take place directly after receiving and storing an image file from the sensor node. A custom function is written so the functionality of uploading an image in storage to the database is concise and can fit anywhere in the system.

```
bool im2Firebase(string im_file) {
  FirebaseData firebasedata;
  String fb_path = "/Gateway-Node";
  bool UPLOADED = 0;
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
  }
  Firebase.begin(FIREBASE_HOST, FIREBASE_AUTH);
  Firebase.reconnectWiFi(true);

  SPIFFS.begin();
  if (Firebase.setFile(firebaseData, StorageType::SPIFFS, fb_path + "/Image", im_file)) {
    UPLOADED = 1;
    SPIFFS.remove(im_file);
    SPIFFS.end();
    return UPLOADED
  }
  else {
    return UPLOADED;
  }
}
```

Figure 51: Image File to Firebase Function

This function shown in Figure 51 takes in the storage path for the image file to be uploaded and returns a boolean value of whether the file is successfully uploaded or not. The code starts by creating a Firebase data object and a database path to store the uploaded file. A boolean value UPLOADED is set to false and continuously attempts to connect to a local Wi-Fi network. Then, the database within Firebase is connected and the flash file system is started. The specific image file is then “set” to the database and if successful, the image file is removed from device storage and the return value is set to true. To have the image files

ready for use as soon as possible, the im2Firebase() function can be called immediately after the data is transferred from the sensor node to the gateway node via BLE.

5.8 Functional Summary

This chapter has discussed all of the individual functions that make up the IoT system created. This section will summarize how these functions interact, how nodes will cooperate, and how data will flow through this single sensor node, single gateway node, and cloud storage system. Figure 52 can be referenced while reading each node's summary.

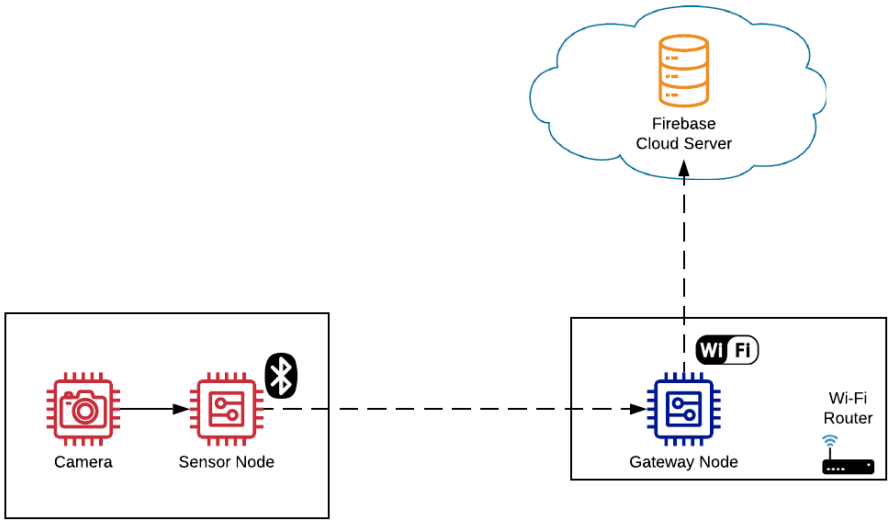

Figure 52: Functional Summary Reference Diagram

SENSOR NODE:

The sensor node gathers bursts of image data using an OV2640 2MP camera module and stores the best quality captured image frame to on-board memory. The node then advertises itself over Bluetooth Low Energy communication channels, hoping to find a suitable client. Once a connection is made, image data chunks are transmitted as BLE notifications to the client. After an entire image frame is sent successfully, the node removes the image file from memory as it is no longer needed. The sensor node now enters deep sleep mode to consume significantly less power until new image data is required. The process is repeated once the node wakes from sleep. A sensor node is assumed to be operating remotely in a crop field on battery power.

GATEWAY NODE:

The gateway node periodically scans its local area checking if there are any BLE devices looking to transfer data. If no such device is found after 20 seconds, the gateway node will enter light sleep for five seconds, then try again. If a device is found, a connection is made, and the gateway receives notifications while storing image data chunks into a file in on-board memory. Once the data transfer is complete and the BLE connection is severed, the BLE controller is disabled before the Wi-Fi hardware is powered on. The gateway then attempts to connect to a known Wi-Fi signal supplied by a nearby router. After Wi-Fi and Firebase Cloud connections are made, a base-64 encoded image file is pushed to the cloud database. The original image file is removed from on-board memory after a successful upload. In case of a failed Wi-Fi connection or unsuccessful file upload, the gateway node is programmed to push all existing files in its memory to Firebase when and while a successful connection is maintained.

FIREBASE STORAGE:

Firebase will securely hold images pushed from the gateway node. Each file push from the gateway is stored as its own base-64 string entry in the database. While possible to manually take this string and download a JPEG image file, a Python Firebase scraper script was developed to automate this. Each time the script is run, all existing image entries on Firebase are saved as JPEG files into a local directory on the user's computer while removing the files from the database. This enables quick cloud file downloads at the press of a button so the images can be used as soon as possible.

Chapter 6

SYSTEM CHARACTERIZATION

This chapter attempts to characterize the system as a whole focusing on power consumption and timing of the functionalities developed in the previous chapter. By collecting this power and timing data, every aspect of the system can be judged based on these two characteristics. Bottlenecks in the system can be identified and potentially fixed while battery lifetime can be extrapolated.

6.1 Functional Power and Energy Testing

This section will analyze the power consumption and timing of each major function described in chapter 5. First, the test setup will be described. Then, the image capture, SD card and SPIFFS file writing, BLE advertising, BLE transmission, and deep sleep will be examined. Following the individual assessments, the sensor node system as a whole will be analyzed with power calculations and battery life estimates. The gateway node's current consumption will be characterized in less detail as it is assumed to be running on a constant power source.

6.1.1 Power Draw Test Setup

The basic idea of measuring power consumption of a system comes down to the simplicity of Ohm's law. The relationship between voltage, current, and resistance is utilized to monitor the voltage drop across a resistor in line with the device. This translates to a system current being drawn and a final delivered voltage to the development board. Figure 53 is a simple diagram depicting these relationships.

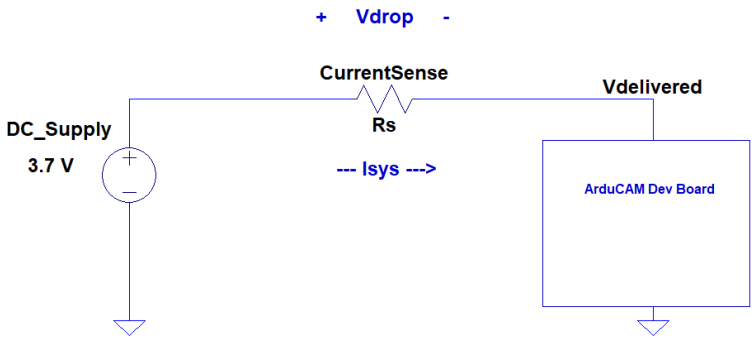

Figure 53: Simple Power Draw Setup Schematic

The system current, I_{sys}, is the ratio of the voltage drop (V_{drop}) across the series resistance to the series resistance itself, R_s. The delivered voltage to the system, $V_{Delivered}$, is the difference between the supply

voltage and the resistor voltage drop. The following equations allow the current draw and the power consumption of the system to be calculated and monitored.

$$\boldsymbol{Eq.1} \quad I_{Sys} = \frac{V_{Drop}}{R_s}$$

$$\boldsymbol{Eq.2} \quad V_{Delivered} = DC_{Supply} - V_{Drop}$$

$$\boldsymbol{Eq.3} \quad P_{Sys} = I_{Sys} * V_{Delivered}$$

In order to make the calculations easy and to make sure the voltage delivered to the development board is enough, the series resistance chosen is 1Ω. This means the system current will ideally be directly proportional to the voltage drop across the resistor; a 100mV drop is equivalent to 100mA drawn by the development board. To obtain time-domain plots of the system current being pulled, an oscilloscope is used to measure the voltage at the high and low sides of the series resistor with one channel each. The math function is used to display the difference of these channels. Figure 54 shows the connections used.

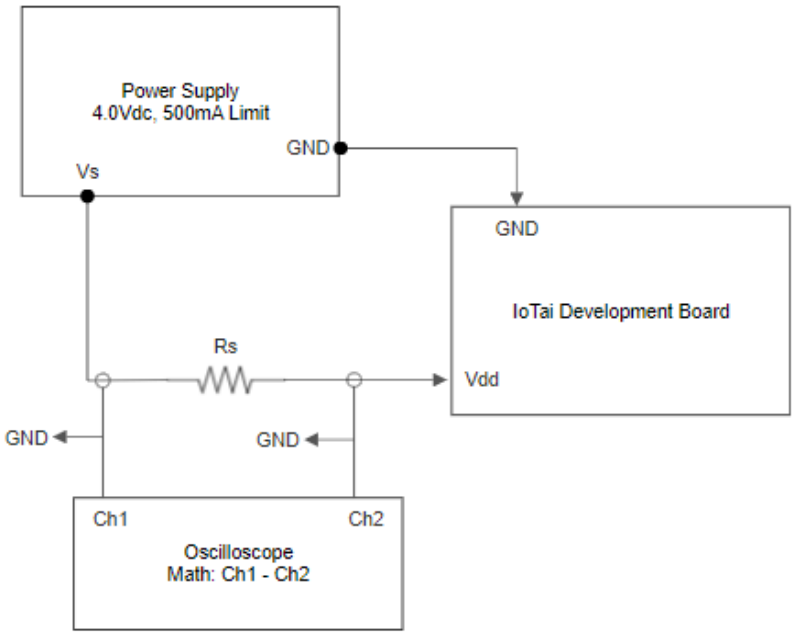

Figure 54: Power Draw Test Connection Setup

The test setup described for measuring current draw of the system is acceptable for all the functionalities other than the deep sleep mode. This is because the oscilloscope being used has voltage resolution down to 1 millivolt per division, which is perfectly fine for any current above a few milliamps and applies to the majority of functions in this system. To accurately measure and plot a current in the

microamp range, an oscilloscope low-noise preamplifier is used. A description of the module and respective test setup is shown in section 6.1.5.

The current draw plots in the following sections were obtained by exporting CSV files from the oscilloscope in the configuration discussed. The files were opened in Excel and the voltage data was converted into current data using Eq.1 shown above. The true value for the series resistor was measured at the time of data collection using a benchtop multimeter: 1.143Ω.

6.1.2 System Idle Power

It is first useful to gauge the system idle current draw, or the current consumed by the development board while not doing anything. A completely void test script was uploaded to the development board and allowed to reach its idle state after booting up. Figure 55 shows this steady-state idle current draw over an elapsed time of one second.

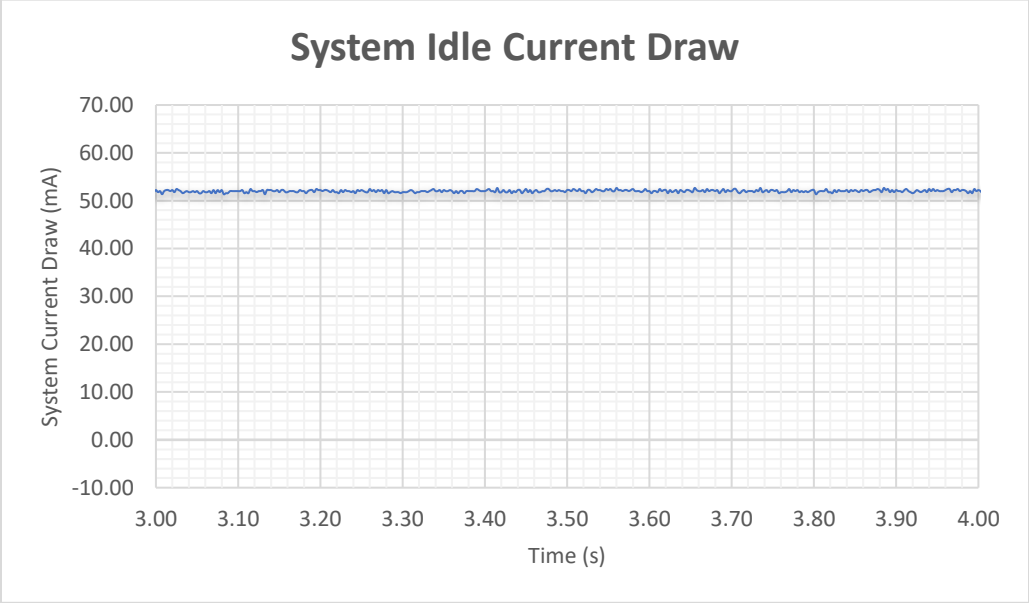

Figure 55: Development Board Idle Current

The average current consumption of the entire development board while doing nothing is 52mA. This allows us to better determine the power usage of significant actions developed for the sensor node. These functions will be “added” onto the existing 52mA idle draw as additional processor tasks are run. The idle current considers powering all electronic components on the development board. This includes the ESP32 chip, OV2640 camera, USB-serial chip, microSD card, and any LEDs or regulators.

This idle current shows the benefit of entering a low-power sleep mode in between processor-heavy tasks. As an example, suppose a typical 2500mAh 3.7V Lithium Ion Polymer battery (20% discharge safety) is powering the board in its idle state shown above. The device would be able to run for approximately only 38 hours before shut-down.

6.1.3 Image Capture Power

The first function developed was image capture using the OV2640. A test script was uploaded to the board to capture a series of five images without storing them. Figure 56 shows the entire script process: booting up, initializing the camera, taking the images, and entering sleep.

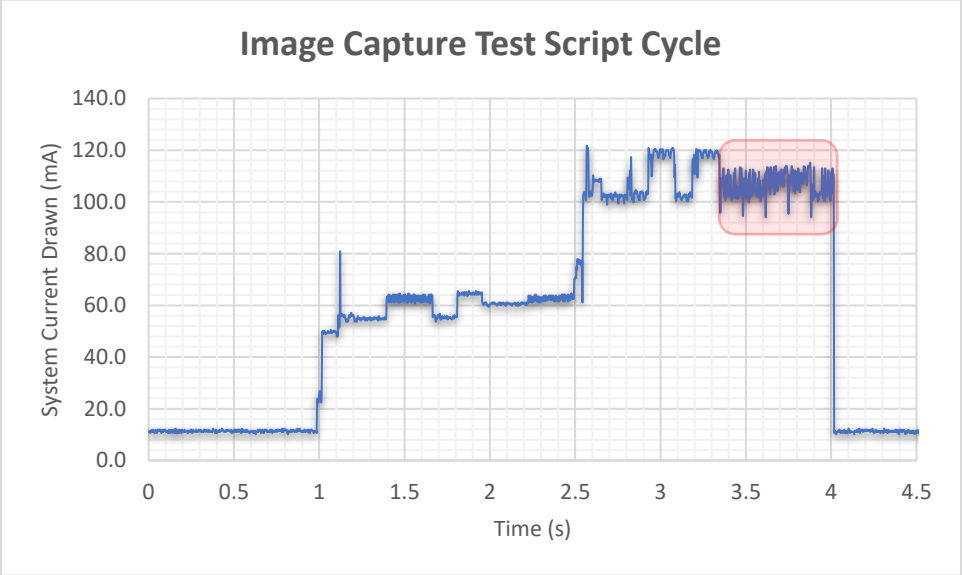

Figure 56: Image Capture Test Script Cycle (Boot to Sleep)

The highlighted portion of the plot shows the system current associated with the series of five images being taken. The time scale on the x-axis is time in seconds taken from the oscilloscope. The highlighted section of Figure 56 is shown in more detail below in Figure 57.

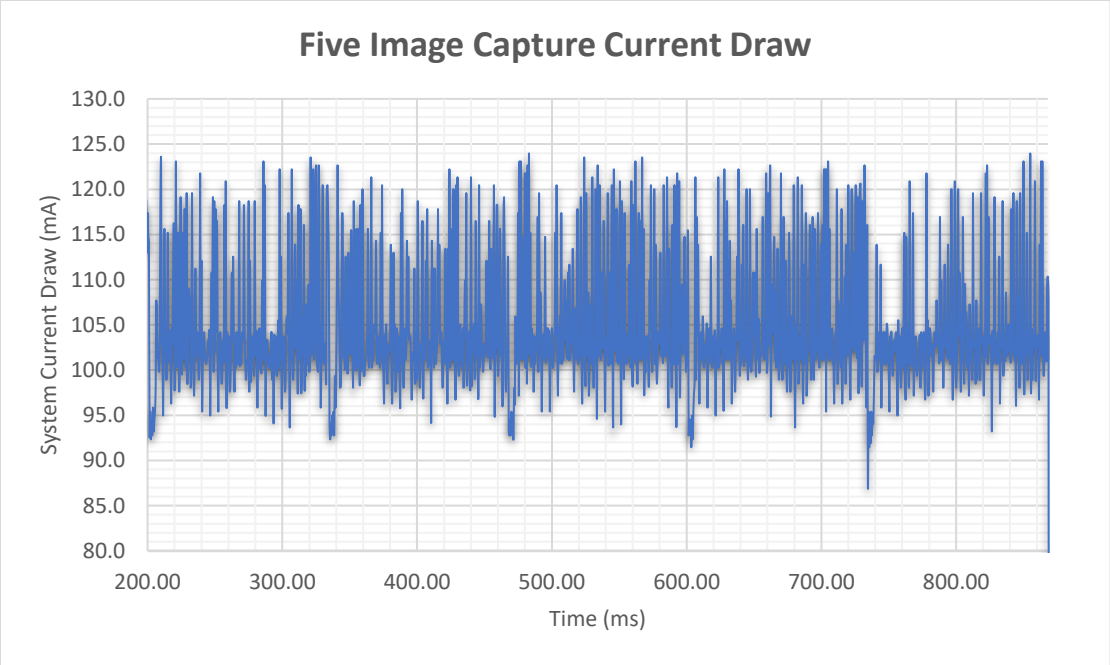

Figure 57: Five Images Captured Test Script

It is easy to discern the periodic nature of the waveform as each image is captured. This plot illustrates both the current drawn and the time taken to capture the series of five 2-megapixel images. From the data captured, the average system current draw over the entire process of images taken is 105mA and takes 660ms to complete. This translates to a time of 132ms per image capture.

6.1.4 SD Card and SPIFFS File Write Power

Next, the current consumed while writing the best image file to storage is analyzed. First, a test script that captures an image, delays for a second, writes the image data to storage (SD or SPIFFS), then delays for another second is uploaded to the board. This test is intended isolate the file writing process and compare the current consumption of each storage type. Figure 58 and Figure 59 show the isolated current draw waveform during both file writes.

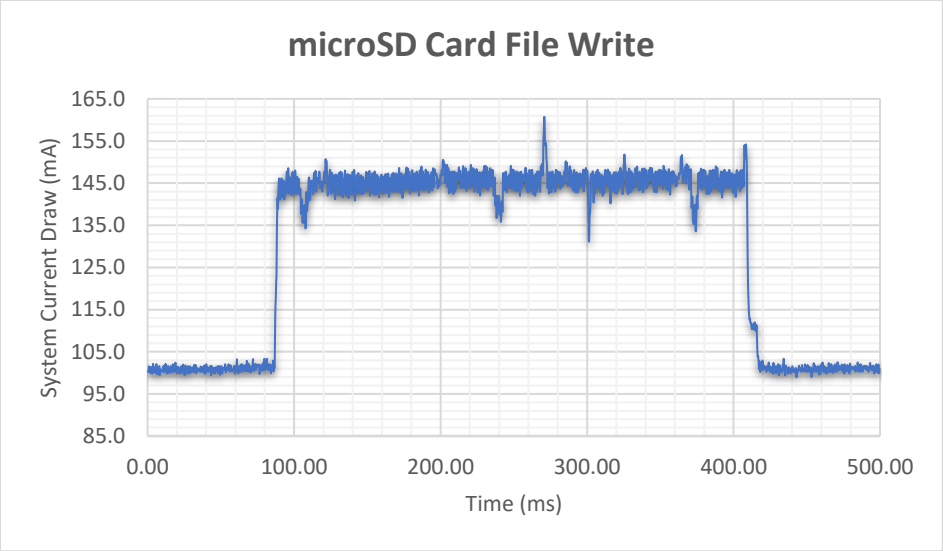

Figure 58: Image File Write Test for microSD

The average system current draw for writing a large file to the SanDisk Industrial microSD card is 146mA. Figure 58 shows a 146kB image file being written in about 0.32s.

A smaller storage alternative to SD cards is the serial peripheral interface flash file system (SPIFFS). The development board being used has 4MB of flash storage. SPIFFS can be used to read and write data files to this chip. While this alternative has much less flexibility in terms of storage size, it uses less idle current than the microSD card and enables a much lower sensor node sleep current. Figure 59 shows the result of the same test as Figure 58, but now writing to flash storage instead.

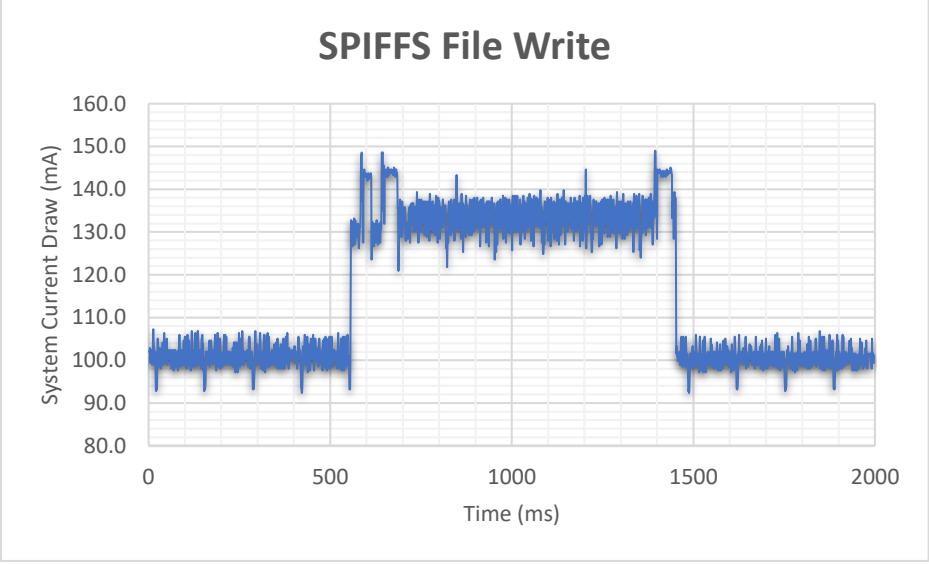

Figure 59: Image File Write Test for SPIFFS

Writing a large image file to flash storage consumes an average current of 134mA but takes about 800ms. To write image files, the microSD card takes less time but consumes more power. At this point, the differences in timing and power consumption are seemingly negligible in choosing between the two storage types.

6.1.5 BLE Server Power

The BLE aspect of the system is the most power hungry and the transmission of the image file is the single most time-intensive process of the sensor node. This section takes a look at the current draw of BLE advertising and BLE transmission current consumption. The energy expended during transmission changes significantly based on the size of the BLE transmit buffer.

The final version of the sensor node code was uploaded to the board as we want to see the BLE current draw as it will happen in our actual system. The sensor node starts advertising its information just after all BLE initialization occurs. Figure 60 and Figure 61 below shows advertising, client connection, and transmission during a test. As a note, the transmission period in the plot also includes the current draw of a continuous storage read (microSD or SPIFFS) while filling the transmit buffer. This combination of processes leads to the largest current draw of the sensor node throughout the image transmission.

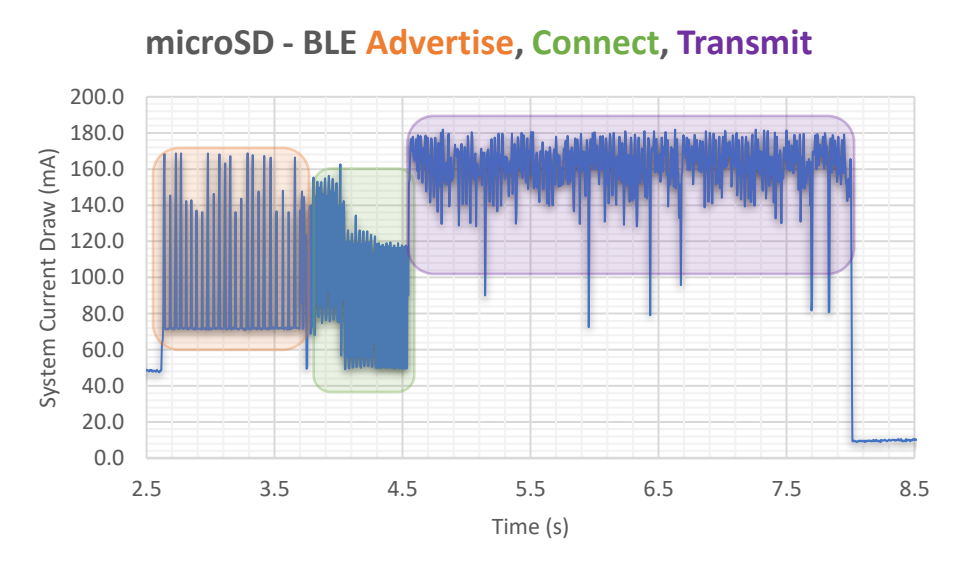

Figure 60: BLE Advertising and Transmission Current Draw for microSD

Figure 60 above shows the timing and current draw of the sensor node during BLE functions using a 23-byte buffer for the Characteristic value. The buffer is filled by reading data from the microSD card. If

the client is already scanning, a BLE connection to the client after the start of advertising typically takes less than 5 seconds, shown in the green and orange section of the figure. The purple section of the figure shows the transmission of a 73kB image, taking approximately 3.5 seconds.

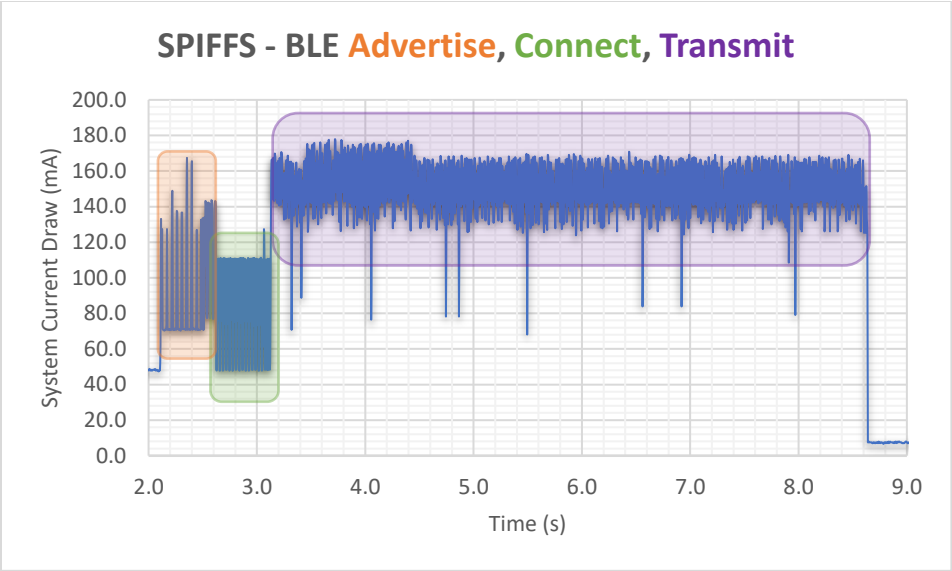

Figure 61: BLE Advertising and Transmission Current Draw for SPIFFS

Figure 61 shows the same information as the previous figure, now while filling the buffer from flash storage. The process shows the transmission of a larger 120kB image file in about 5 seconds.

As the size of the buffer used is the biggest factor of image file transmission throughput, testing was done using different sized buffers to illustrate this significant effect while transmitting the same 110kB image file using both storage types. Table 3 summarizes the findings.

Table 3: Variable Tx Buffer Size Effect on 110kB File Throughput

BLE Buffer Size	1 Byte	4 Bytes	8 Bytes	16 Bytes	23 Bytes
File Tx Time (microSD)	~81s	~20s	~11s	~6s	~4s
File Tx Time (SPIFFS)	~120s	~30s	~16s	~7s	~4s

As the transmit buffer size increases to its limit of 23 bytes, the transmission time decreases considerably for the same image file. From this table, we can conclude that using the 23-byte buffer will minimize the image file transmission time. When the buffer is at its optimal size, the transmission time of each storage type becomes similar. By minimizing the transmission completion time, the sensor node saves

valuable energy. At the optimal size, the data transmission throughput of a sensor node is about 27.5kBps, or 220kbps.

Next, we can characterize both the advertising and transmission current consumption individually by isolating the data. First, the advertising is analyzed as it is independent from the storage type used. Figure 62 shows BLE advertising over a 115ms time span.

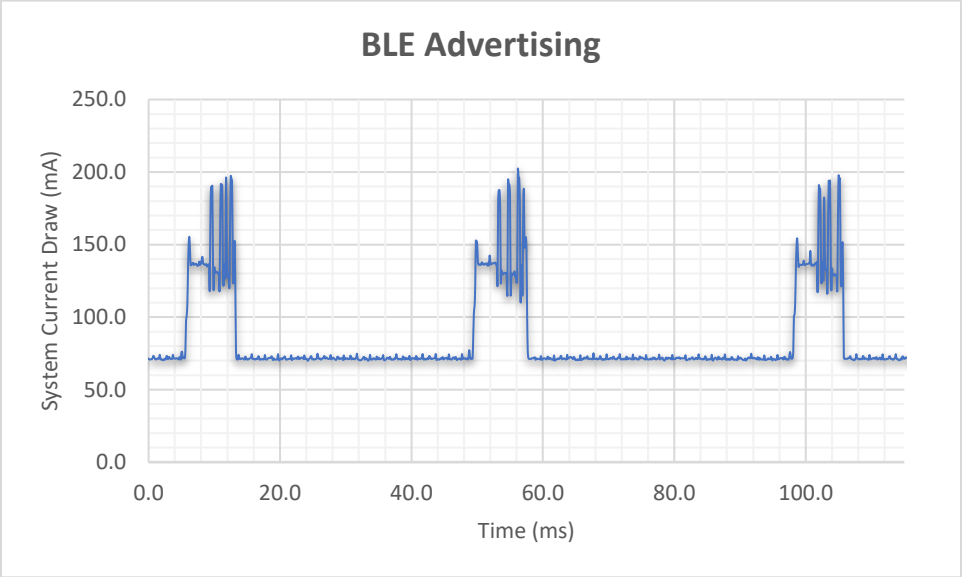

Figure 62: BLE Advertising Current Draw Plot

The advertising current consumption over this time period hovers around 83mA. The BLE protocol saves energy by periodically advertising itself in bursts while turning off RF hardware in between. This advertising scheme is the default on the ESP32 and has a period of 45ms with an advertising window of 5ms.

Next, the BLE transmission current consumption using both SPIFFS and microSD card reading to fill buffers is characterized. The test script uploaded to the board for Figure 63 and Figure 64 continually read data from the same image file into the optimal 23-byte buffer and send BLE notifications until the end of the file. The test image file used was 73kB.

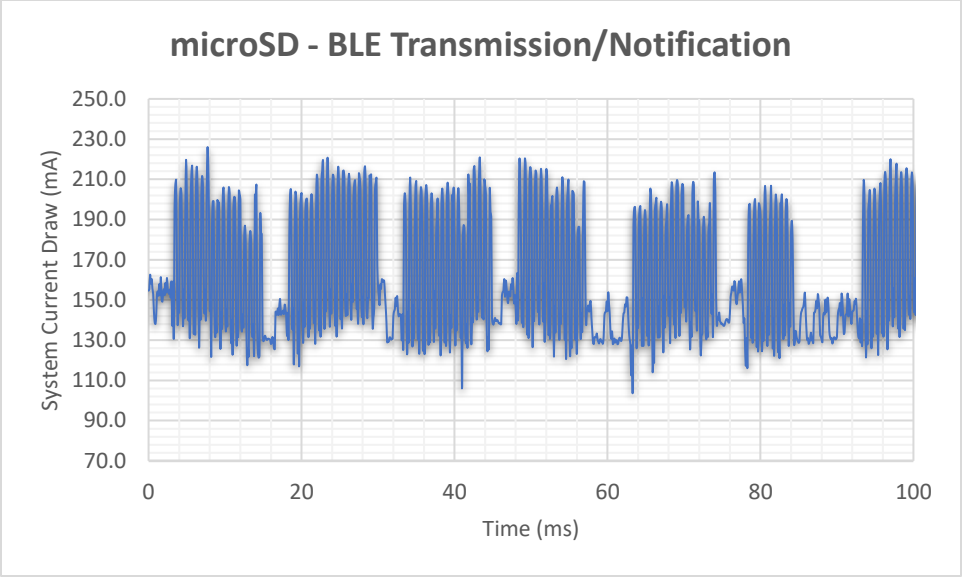

Figure 63: BLE Transmission Current Draw Plot using microSD

Figure 63 shows a typical transmission current waveform over the span of 100ms while reading from the microSD to fill the transmit buffer. Over the entire file transmission, the average current draw is 162mA. Figure 64 shows an identical test but while using SPIFFS to store images and to fill the transmit buffer.

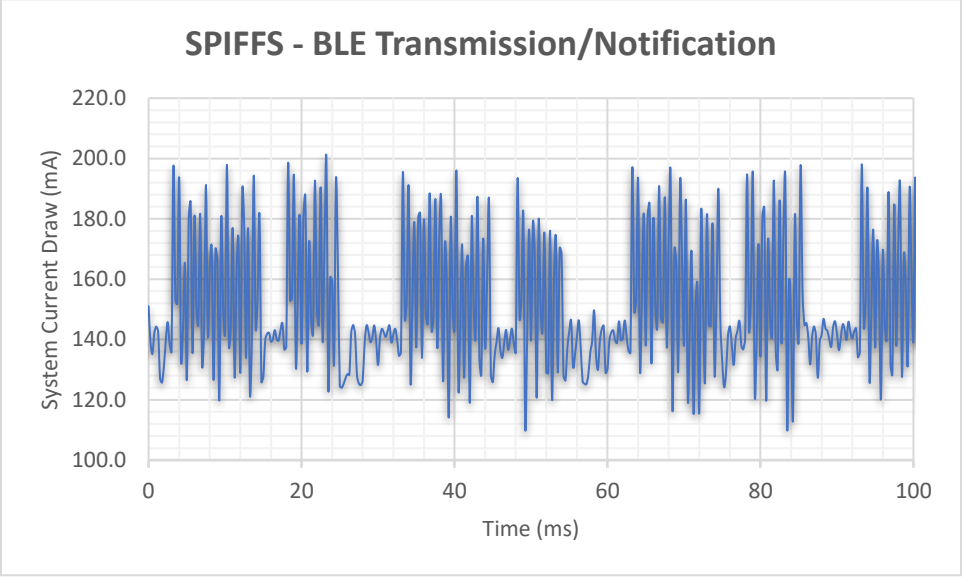

Figure 64: BLE Transmission Current Draw Plot using SPIFFS

The waveform in Figure 64 shows that SPIFFS consumes an average of 150mA to continuously read and transmit data. As seen in Table 3, the time it takes to transmit an image file using the optimal buffer

size is about the same for either storage type. Both Figure 63 and Figure 64show that using SPIFFS versus the microSD card uses a slightly smaller amount of current. While the two storage methods compared in this work have varied slightly in speed and current consumption, the most dramatic effect of using one over the other comes while the sensor node is in deep sleep.

6.1.6 Sleep Mode Power

To obtain accurate measurements for sleep mode, an oscilloscope preamplifier was used to achieve large differential signal gain. The preamplifier used was the LNA10 by AlphaLab, shown in Figure 65. By using this, sub-microvolt signals can be displayed on scopes which typically only go down to 1mV/division on the vertical axis [29]. The tool included a differential input with a tunable analog low-pass filter, so my testing configuration was identical. The LNA10 was used with a gain of 1000.

Figure 65: AlphaLab LNA10

A test script is configured so the ESP32 goes into deep sleep for 10 seconds, reboots, and repeats. This will illustrate the low-power current for the SoC on this particular development board. The test is first run on a development board with a microSD card inserted and then run again with no card inserted. These two test runs show how the storage type used can have a drastic effect on the standby current draw of this system. As mentioned in chapter 5, the microSD card draws a considerable amount of current while in idle.

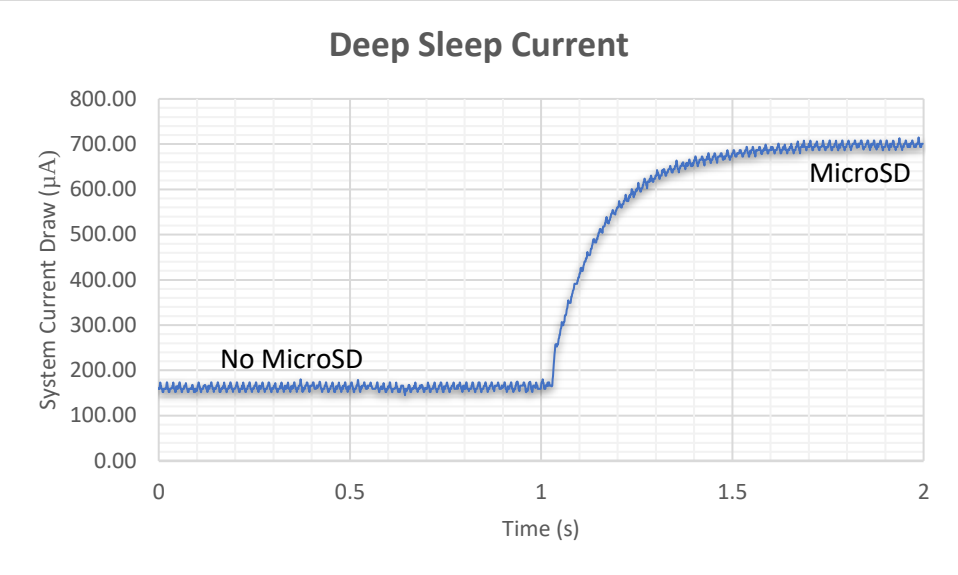

Figure 66: Deep Sleep Current Draw without and with MicroSD

Using flash storage with no SD card inserted, the development board draws an average current of 162µA. With the SD card inserted, the current rises to an average of 700µA. Figure 66 shows that the idle current draw of the microSD consumes about 77% of the deep sleep power of the development board at 538µA. As standby current is the most important low-power aspect of the system in question, and therefore it is recommended that the on-board flash storage be used for maximum sensor device lifetime.

6.2 Sensor Node Power Characterization – Optimal Configuration

To characterize the sensor node system's power, the data collected from the Section 6.1 is gathered. This section aims to characterize the most power-efficient version of the sensor node, so the buffer size is set to 23-bytes and flash storage is used. Two node states are defined: active and standby. The active state is where the CPU is running; the standby state is where the CPU is put to sleep.

Table 4 shows the power and time consumption of the most major actions in the system while in its active state. The buffer size is set to the optimal 23 bytes during transmission and measured time shows a conservative estimate of the action's duration. The power column is based on the fact that the system runs on a battery voltage of 3.7 volts. The BLE transmission time will vary based on image file size but the largest files captured (~200kB) were successfully transmitted in about 10 seconds. The maximum duration is set to 10 seconds to be ultra-conservative, even though the majority of images were compressed to a size less than 200kB.

Table 4: Sensor Node Active Mode - Optimal

State	Duration (Max)	Avg. Current (SPIFFS)	Power	Percent ON
Boot Initialization	2.5s	60mA	222mW	13.5%
Image Capture	1.0s	90.13mA	333mW	5.4%
BLE Advertising	5.0s	83.0mA	307mW	27.0%
BLE Transmission	10.0s	150mA	555mW	54.1%

The table shows the sensor node functions and the corresponding durations, average currents, and the percent of time each function is on during the active state. The conservative estimate of the total duration of the active state is 18.5 seconds. This active state will be turned on every time a new image acquisition takes place.

In between active states, the sensor node will go into sleep, or its standby state. The power consumed in this state is more important to overall device lifetime than the active state power as the sensor node will spend the overwhelming majority of its time here. The standby state is summarized in Table 5.

Table 5: Sensor Node Standby Mode - Optimal

State	Duration	Avg. Current (SPIFFS)	Power
Deep Sleep	Varies	162µA	599µW

Based on the analysis done in section 6.1.6, the deep sleep current for the sensor node using flash storage is about 162µA. The duration of time the node spends in its standby state will vary but will always be well above 99% of the total standby plus active time. The image acquisition frequency will be very low, likely multiple time a day. Figure 67 below shows the percentage of time on for active and standby functions over a duration of one hour as a pie chart. This helps visualize the importance of standby current.

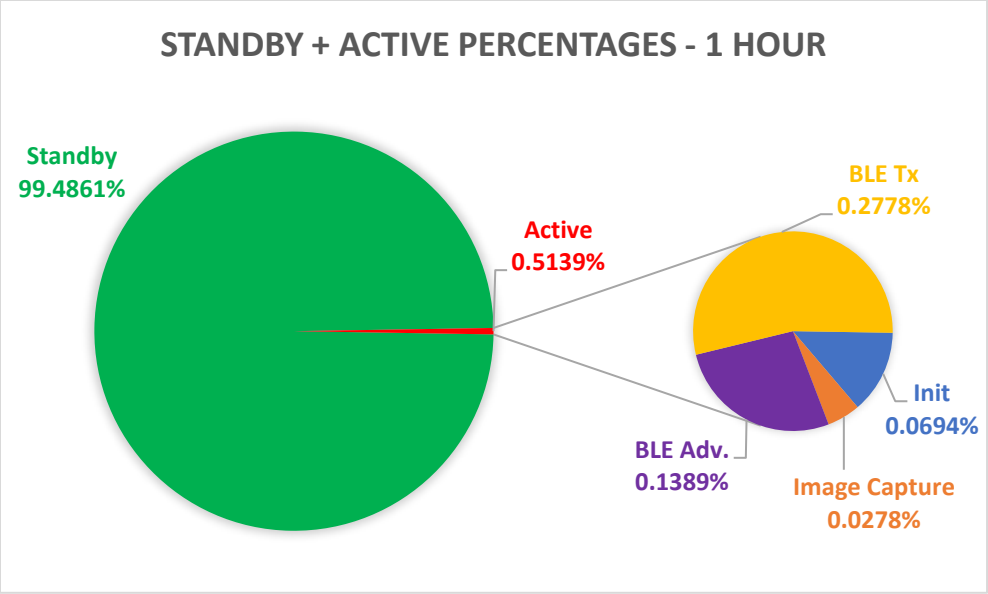

Figure 67: Standby and Active Mode Percentages for One Hour

Figure 67 illustrates that over one hour, the sensor node would spend 99.486% of its time in standby mode. The entire active mode accounts for 0.514% of time. If the image capture frequency stays at anything less than 24 captures per day, the sensor node will easily stay in standby mode over 99% of the time. Using this data, the next section will estimate battery life for various image acquisition frequencies and battery capacities.

6.3 Battery Life Estimates

First, the sensor node's active state functions are characterized by how many individual executions per battery they could maintain. Table 6 summarizes these findings. The battery options chosen are all Lithium Ion Polymer (LiPo) battery packs with sizeable capacities and manageable dimensions for this application. LiPo batteries are lightweight, have impressive capacities, and high discharge rates to supply more current to their load [30]. All are 3.7-volt batteries with 2-pin JST-PH connectors and built-in protection circuitry. The connectors fit with the development board and the voltage is perfect to power the ESP32.

Table 6: Executions Per Battery for Sensor Node Functions

Function	Time	Avg. Current	Executions per 2500mAh Battery	Executions per 4400mAh Battery	Executions per 6600mAh Battery
Boot Init	2.5s	60mA	60,000	105,600	158,400
Image Capture	1s	90.13mA	99,855	175,746	263,619
BLE Adv.	5s	83mA	21,686	38,168	57,253
BLE Tx	<10s	150mA	6,000	10,560	15,840
Deep Sleep	N/A	162µA	643 Days Continuous	1131 Days Continuous	1697 Days Continuous

Three realistic acquisition frequency scenarios are considered with three different battery capacity options to extrapolate measured current consumption data into battery lifetimes for the sensor node. Scenarios will be very conservative and will use a current of 150mA for the entirety of its active state and a current of 162µA during its standby state. The conservative active estimate is meant to be an absolute worst-case scenario. Example scenarios are listed below.

Scenario 1: Image Acquired One Time per Day

Scenario 2: Images Acquired Five Times per Day

Scenario 3: Images Acquired Every Other Day

Table 7 gives conservative battery life estimates based on battery capacity and each scenario. The calculations in Table 7 were done using a 20% battery discharge. Batteries are seldom fully discharged, so manufacturers often use the 80% depth-of-discharge formula to rate a battery. This means that only 80% of the available energy is delivered and 20% remains in reserve. Manufacturers argue that this is closer to a field representation than using a full cycle [31].

Table 7: Sensor Node Conservative Lifetime Estimates with Various LiPo Batteries

LiPo Battery Capacity (20% Discharge)	Scenario 1 1 Image/Day	Scenario 2 5 Images/Day	Scenario 3 0.5 Images/Day
2500mAh 47 x 61 x 6.7 (mm) 43g $15	429 days	258 days	468 days
4400mAh 69 x 37 x 18 (mm) 95g $20	755 days	454 days	824 days
6600mAh 69 x 54 x 18 (mm) 155g $30	1133 days	682 days	1236 days

Table 7 shows the minimum amount of time the sensor node will last on a specific battery capacity using three realistic acquisition frequencies. In 8 out of the 9 scenarios, the sensor node will last well over a year on a single battery charge. The application does not strictly limit the size or weight of the battery used, but this information is included in the table for reference, as well as approximate costs.

The sensor node with an optimal configuration as described in this section achieves an BLE transmission data-rate of 220kbps. The node spends well over 99% of its life in standby mode consuming 162µA of current. The active period of the node takes 18.5 seconds at a maximum, drawing a peak current of 150mA during transmission. Using a 6600mAh battery while capturing and transmitting 5 images per day, the sensor node will last at least 682 days.

6.4 Gateway Node Power Characterization

This section shows the gateway node's current draw during its primary functions: BLE scanning, BLE reception, and Wi-Fi transmission to the cloud. As this node is not assumed to be running on battery power, the discussion is brief.

6.4.1 BLE Client Power

The gateway node performs two functions that the sensor node does not. BLE scanning and BLE data reception. This section shows the current consumption of the development board during these two actions individually.

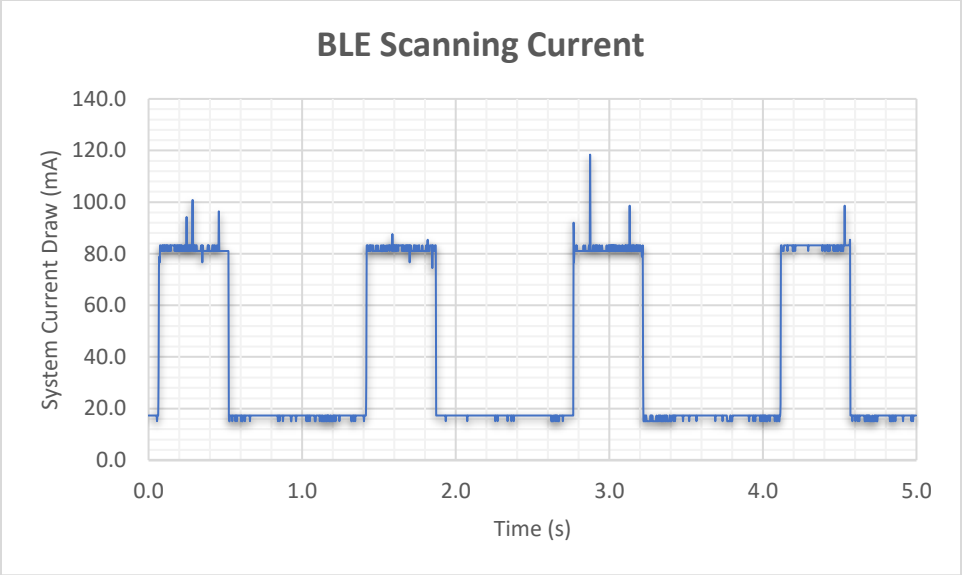

Figure 68: BLE Scanning Current Test

The test results in Figure 68 show BLE scanning over five seconds draws an average of 40.7mA. The scanning interval is 1.35 seconds long and the scanning window is 0.45 seconds, as set in the code. Over the scanning window, the system consumes a maximum current of about 84mA.

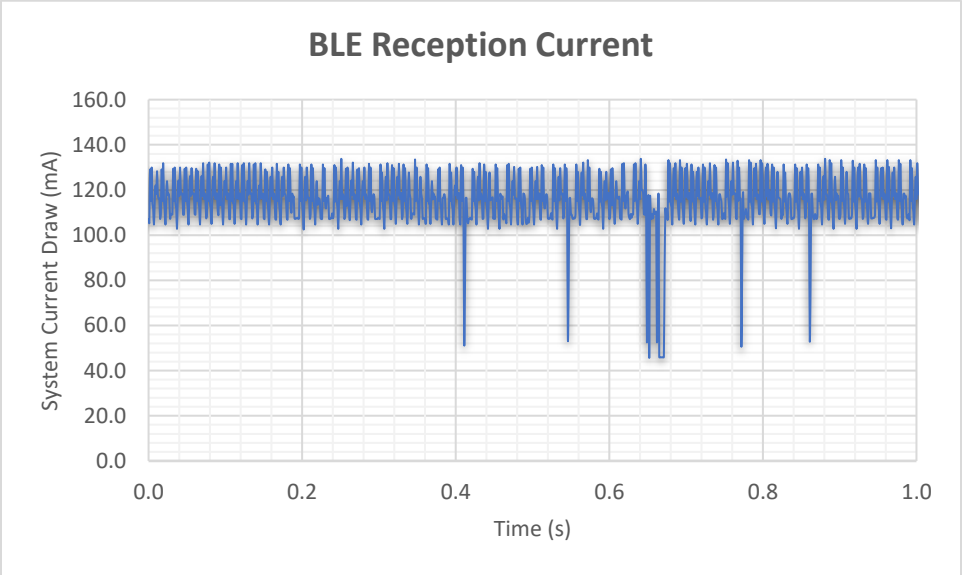

Figure 69: BLE Reception Current Test

As the gateway node receives data sent from the sensor node, the current drawn averages 120mA, seen in Figure 69. This number is slightly lower than the BLE transmission current of 150mA.

6.4.2 Wi-Fi to Cloud Power

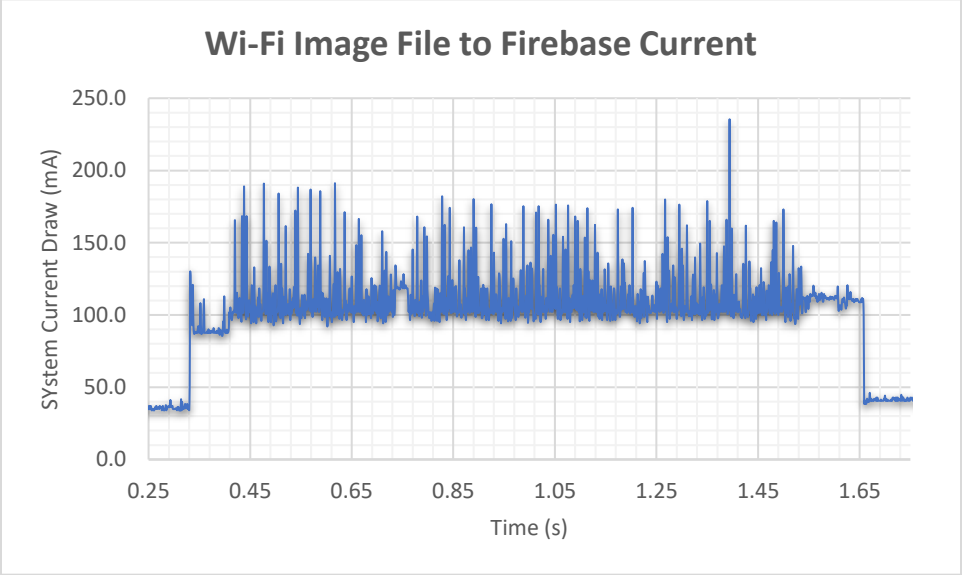

Figure 70: Wi-Fi Image File to Firebase Cloud Current Test

While uploading the image file to Firebase, the development board consumes an average of 112mA of current. In the test shown in Figure 70, an 84.5kB JPEG image is first encoded into base-64, resulting in a 115.3kB file. This file is uploaded to Firebase in 1.3 seconds. This upload data rate sits at 709.5kbps.

6.5 Extra Low-Power Design Methods

As with any development board used for prototyping, there are certain electrical components that are not needed during actual deployment but have high quiescent or idle currents. These could include indication LED's, USB-to-UART transceivers, or high quiescent current linear voltage regulators.

This development board has four LED's to indicate power, charge, and programming, while the last is software controllable. In a deployed system, none of these are used but may contribute to system current draw in active and standby states. If these surface-mount LED's are turned on, they can pull around 10mA for each LED. If all are on, this could contribute 40mA to the system's current. Even when these LED's are turned off (like in deep sleep mode), they can leak current in the microamp range. This is especially true when the LED is controlled by a CMOS GPIO line that can rest at anywhere less than 0.4V in its off state. Even single microamps of current being leaked can affect the device lifetime duration significantly, so it is best to completely remove these LED's from the board altogether.

USB-to-UART transceivers used on development boards help to upload program data easily and quickly to memory via a USB port. This functionality is not needed after the board has been programmed and deployed. The development board in this thesis uses the classic Silicon Labs CP2102 USB to UART bridge IC. When this particular chip is suspended (not in use), the typical supply current draw is 80µA [32]. If this chip is powered on a development board when using battery power, removing it could significantly elongate device lifetime. The ArduCAM development board fortunately switches the USB signal chain completely off when powered by a battery, so this does not add to its sleep current when in standby.

The majority of this development board's sleep current comes from a high quiescent current low-dropout regulator (LDO). This board specifically uses the NCP500 low-noise, low-dropout voltage regulator to power all 3.3V systems, including the microSD card reader and the ESP32. According to the datasheet, this chip has a typical quiescent current of 195µA in its specific configuration [33]. As this sleeping system pulls a total of 162µA, the quiescent current of this chip is less than its typical value but is likely responsible for the majority of the sleeping current of the system. It is entirely possible to replace this TSOP-5 chip with another that has an ultra-low quiescent current but similar current sourcing abilities. For example, the NCP703 is an ultra-low noise LDO able to source a maximum of 750mA with an ultra-low quiescent current of 12µA [34]. By changing out the voltage regulator, the sleeping current of this system could be reduced drastically, enabling many-year battery lifetimes.

6.6 IoT System Comparison

A comparison of power information is provided in this section. The system developed in this thesis is not directly comparable to the systems reviewed in Chapter 2, but the data is useful, nonetheless. Table 8 shows a comparison of aspects of the systems reviewed that were able to be found. The third paper reviewed in Chapter 2 was not included in the table as the report did not include any power information.

Table 8: IoT System Specifications Comparison

System	Sensor Node SoC	Tx Current	Sleep Current	Tx Data Size	Data Rate
IoTai Dev.	ESP32	150mA (BLE 4.2)	162µA (Dev.)	50-250 kB	220kbps
Vinduino	LM-210	120mA (LoRa)	Unknown	10-20 Bytes	810bps
Multi-Hop	BMD-340	32.6mA (BLE 5.0)	3.65µA (PCB)	21-29 Bytes	7.2kbps

It is likely that significant differences in transmission and sleep currents are associated with hardware variations. The Vinduino and Multi-Hop systems were developed on optimized custom PCB's with surface mount SoC's. This system was characterized by monitoring power of the entire development board.

The Multi-Hop IoT system was optimized for isolated monitoring of short sensor data values. The system is incredibly efficient as their custom sensor node PCB consumed 32.6mA while transmitting packets and 3.65µA while sleeping. The node is able to achieve such low power consumption by operating at 1.8V and only transmitting very small data messages less than 30 bytes. The maximum data rate achieved by this system is 7.2kbps. The Vinduino system uses LoRa to transmit small data packets long distances. Even though the data throughput achieved is a lowly 810bps, the system still achieves a low energy consumption as the packets are 10 to 20 bytes. These systems can be classified as isolated, small-data, low-throughput IoT networks. By having loose requirements for data rate and data packet size, the systems are able to sacrifice these metrics for lower power.

The system developed in this thesis qualifies as an isolated, big-data, high-throughput IoT network. The requirement for low power still exists while the requirements for data rate and data size are much more restricted, needing to be maximized. While these systems compared do not have the same objective, the system designed in this work helps show that IoT systems do not have to be contained to the periodic collection and dissemination of small amounts of data. Isolated networks can still operate on batteries and last for years while sensing and collecting large quantities of very diverse data and transmit at high data rates.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Summary and Conclusion

This thesis proposed, developed, and characterized an IoT prototype system to be used as the data acquisition step in a computer vision and machine learning pipeline for predicting the timing and yield of crop harvests. The parts of the network developed were the sensor node, the gateway node, the cloud server, and the wireless link between all three. The nodes are based on the ESP32 SoC and tested with the ArduCAM IoTai development board. The sensor node and gateway node communicate between each other using Bluetooth Low Energy technology. Full colored 2-megapixel images are captured and compressed using JPEG by the sensor node. These compressed image files are kept in flash storage until a connection is made between the sensor and gateway. Once image data has been received, the gateway node uses Wi-Fi to push data files to a real time database using Firebase cloud storage. Once in the cloud, the images are easily able to be downloaded and used by a computer vision and machine learning pipeline.

This prototype system was developed to remedy specific shortcomings from previous methods, namely: inconsistencies of acquisition frequency, varying illumination levels, varying viewing angles and heights, as well as a time-consuming collection process. An IoT system as proposed can be programmed to acquire crop images at any set frequency, even at specific times of day (morning, midday, and evening for example). This eliminates time inconsistencies from manual image collection caused by human error or forgetfulness and creates a uniform dataset for complex processing models to analyze. By fixing exact times to capture crop images, the problem of varying illumination levels caused by time of day is mitigated. When the sensor node is physically deployed, it will stay in a single stationary location to monitor a specific area of crops. This alleviates irregularity produced by varying viewing angles and heights caused by human image collection. The sensor node is designed to be as low power as possible to extend battery lifetime. This extension makes this system almost fully autonomous and only requires human input every year or more to change batteries. Compared to manual human image capture, this system could easily save an estimated few hours in time per day for a farmer.

The IoT system characterized meets the goals and requirements for the agriculture-specific network but is not limited to this industry. This network can be utilized by any application that requires a wireless

sensor network (WSN), high data rates, low power consumption, short range communication, and large amounts of data to be transmitted at low frequency intervals. As IoT trends towards the inclusion of edge computing and computer vision or machine learning and away from being their own standalone systems, these characteristics will become increasingly desirable. Crop monitoring and yield prediction systems are a prime example of this class of system, drifting further away from the typical low data rate, low bandwidth IoT systems of the past.

Important specifications measured with the prototype system on a development board are now summarized. The IoT network achieves preliminary goals initially set for this work: gateway nodes cost less than $30 and sensor nodes will last at least a year on battery power. The sensor nodes will cost closer to $45 with the inclusion of LiPo batteries chosen. With the use of a fully integrated low-cost hardware development board, a free software development environment, and a free cloud storage, each node's cost is minimized. During its active state, the sensor node consumes a maximum average current of 150mA while transmitting over BLE. The node spends over 99% of its time in standby mode consuming 162µA. In a scenario of capturing and transmitting five images per day, a sensor node could be powered from a single 6600mAh LiPo battery for an impressive 682 days. Over this lifetime the sensor node could capture and upload 3,410 images to Firebase. The system also achieves impressive WSN data throughput. Node-to-node BLE communication reaches 220kbps and node-to-cloud Wi-Fi communication reaches 709.5kbps. The system underwent multiple end-to-end system tests to verify complete process operation. The system functions performed perfectly in tests up to 20 meters between sensor and gateway; no tests exceeding this distance were done.

7.2 Future Considerations and Recommendations

This system contains many attractive elements and serves as a well-performing standalone IoT network to monitor a single crop area. This thesis was developed with scalability in mind and also serves as an excellent starting point to create a many-node IoT mesh network to have full coverage of a large area of interest. The following ideas are presented in chronological order of future development.

An idea to provide a wider reach for the two-node, point-to-point system developed in this thesis is to make the single sensor node mobile. If a single sensor node were mounted on a rover, for example, the rover could be programmed to drive to designated spots in a crop field where the sensor node would carry

out its image collection process as developed in this thesis. Instead of immediately transmitting data off-board, the device would store it to memory. Once the rover had driven to all designated locations, it would make its way back to within range of the gateway node, enabling the sensor node to dump its collected data. The gateway would proceed with its functionality and upload the image files to Firebase. This is a less intensive method than a small-area mesh network and could produce similar results. The gateway node's firmware would be unaltered, and the sensor node's firmware would only need slight modification. This same idea could be applied to a sort of long railway with a moving part holding the sensor node. There are adapter boards available that can hold up to four equivalent cameras at once, extending the utility of this idea.

Next, it would be beneficial to design and manufacture a custom sensor node PCB with an emphasis on the inclusion of power efficient supporting electronics. A board that integrated a barebones SoC with a low quiescent current LDO and clever software (GPIO) controlled peripheral power chains for a camera and memory card could result in an ultra-low power node with the same WSN capabilities described in this thesis. If the design of an entire PCB is too complicated or time-intensive, modifying the existing development board would be simple and very effective, too. Removing any LED's and replacing the old LDO with one that has the same pin-out, package size, and current sourcing ability but much lower quiescent current could drastically improve the system sleeping current.

To ensure this system could cover a large area, the number of sensor nodes must increase. This could enable the network to reach further distances because all sensor nodes could act both as collectors and repeaters. The data could hop from one node to the next, eventually reaching the gateway. This idea is known as a mesh network, illustrated in Figure 71. Sensors are depicted as orange and the gateway as grey.

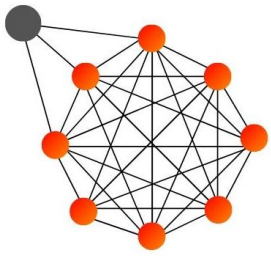

Figure 71: IoT Mesh Network Topology

In this topology, each node has at least two ways to send and receive information. This ensures the whole system does not rely on one node only. Creating a network routing and scheduling scheme for a BLE mesh network using the hardware and software functionality developed in this work could act as an entire thesis topic. The network scheme could use Shortest Path Bridging (SPB) that allows information to be transferred by the shortest available path of nodes. SPB is specified in the IEEE 802.1aq standard. The BLE communication range depends on many things including transmit power, physical environment, and receiver sensitivity. Typically, the realizable range for BLE is between 10 and 100 meters for outdoor applications. A mesh network with sensor nodes that are within this range of each other could span the area of an entire crop field. The ESP32 SoC contains a BLE Mesh core optimized for creating large-scale device networks to get started with this application. A correctly implemented mesh network using the developed sensor and gateway nodes would efficiently enable constant monitoring and imaging of entire farms.

WORKS CITED

[1] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. H. D. N. Hindia, "An Overview of Internet of Things (IoT) and Data Analytics in Agriculture: Benefits and Challenges," *IEEE Internet Things J.*, vol. 5, no. 5, pp. 3758–3773, Oct. 2018, doi: 10.1109/JIOT.2018.2844296.

[2] A. M. Apitz, "Towards a Strawberry Harvest Prediction System Using Computer Vision and Pattern Recognition," California Polytechnic State University, San Luis Obispo, California, 2018.

[3] Y. Fitter, "Strawberry Detection Under Various Harvest Stages," California Polytechnic State University, San Luis Obispo, California, 2019.

[4] "Breaking the curse of small data sets in Machine Learning: Part 2." https://towardsdatascience.com/breaking-the-curse-of-small-data-sets-in-machine-learning-part-2-894aa45277f4 (accessed May 03, 2020).

[5] "Agricultural IoT will see a very rapid growth over the next 10 years." https://machinaresearch.com/news/agricultural-iot-will-see-a-very-rapid-growth-over-the-next-10-years/ (accessed Apr. 18, 2020).

[6] R. van der Lee, "Vinduino, a wine grower's water saving project." https://hackaday.io/project/6444-vinduino-a-wine-growers-water-saving-project (accessed Mar. 25, 2020).

[7] R. A. Kjellby, L. R. Cenkeramaddi, A. Froytlog, B. B. Lozano, J. Soumya, and M. Bhange, "Long-range & Self-powered IoT Devices for Agriculture & Aquaponics Based on Multi-hop Topology," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, Limerick, Ireland, Apr. 2019, pp. 545–549, doi: 10.1109/WF-IoT.2019.8767196.

[8] M. Manideep, R. Thukaram, and S. M, "Smart Agriculture Farming with Image Capturing Module," in *2019 Global Conference for Advancement in Technology (GCAT)*, Oct. 2019, pp. 1–5, doi: 10.1109/GCAT47503.2019.8978368.

[9] F. Lutz and M. Korsloot, "Introduction Alcom Electronics," p. 18.

[10] A. H. Jebril, A. Sali, A. Ismail, and M. F. A. Rasid, "Overcoming Limitations of LoRa Physical Layer in Image Transmission," *Sensors*, vol. 18, no. 10, Sep. 2018, doi: 10.3390/s18103257.

[11] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia, and T. Watteyne, "Understanding the limits of LoRaWAN," *ArXiv160708011 Cs*, Feb. 2017, Accessed: Jan. 16, 2020. [Online]. Available: http://arxiv.org/abs/1607.08011.

[12] "Develop with LoRa for low-rate, long-range IoT applications," *www.mwee.com*, Jan. 10, 2018. https://www.mwee.com/design-center/develop-lora-low-rate-long-range-iot-applications (accessed Mar. 25, 2020).

[13] "LoRa Gateway." https://www.vinduino.com/portfolio-view/lora-gateway/ (accessed Mar. 25, 2020).

[14] B. Ray, "What Is SigFox?" https://www.link-labs.com/blog/what-is-sigfox (accessed Jan. 16, 2020).

[15] “Advantages of SigFox | Disadvantages of SigFox.” https://www.rfwireless-world.com/Tutorials/advantages-and-disadvantages-of-Sigfox-wireless-technology.html (accessed Apr. 18, 2020).

[16] B. Ray, “A Bluetooth & ZigBee Comparison For IoT Applications.” https://www.link-labs.com/blog/bluetooth-zigbee-comparison (accessed Apr. 18, 2020).

[17] B. Ray, “ZigBee Vs. Bluetooth: A Use Case With Range Calculations.” https://www.link-labs.com/blog/zigbee-vs-bluetooth (accessed Apr. 18, 2020).

[18] “Arduino Nano 33 BLE | Arduino Official Store.” https://store.arduino.cc/usa/nano-33-ble (accessed Apr. 18, 2020).

[19] “Particle.” https://docs.particle.io/argon/ (accessed Apr. 29, 2020).

[20] “esp32_datasheet_en.pdf.” Accessed: Feb. 01, 2020. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[21] L. Jackson, “Introducing Arducam IoTai - The Ultimate IoT (Internet of Things) Board with Camera Support, Based on ESP32 and in the Shape of Arduino UNO,” *Arducam*, Aug. 09, 2019. https://www.arducam.com/arducam-iotai-esp32-camera-module-arduino-uno-r3-board/ (accessed Apr. 29, 2020).

[22] “OV2640DS.pdf.” Accessed: May 03, 2020. [Online]. Available: https://www.uctronics.com/download/cam_module/OV2640DS.pdf.

[23] B. Chung, “SanDisk Industrial microSD card Datasheet,” p. 29, 2016.

[24] M. Afaneh, “Bluetooth Low Energy: A Primer,” *Interrupt*, Jul. 30, 2019. https://interrupt.memfault.com/blog/bluetooth-low-energy-a-primer (accessed May 09, 2020).

[25] “ESP32 Bluetooth Low Energy (BLE) on Arduino IDE,” *Random Nerd Tutorials*, May 16, 2019. https://randomnerdtutorials.com/esp32-bluetooth-low-energy-ble-arduino-ide/ (accessed May 09, 2020).

[26] C. Coleman, “A Practical Guide to BLE Throughput,” *Interrupt*, Sep. 24, 2019. https://interrupt.memfault.com/blog/ble-throughput-primer (accessed May 09, 2020).

[27] “Core Specifications,” *Bluetooth® Technology Website*. https://www.bluetooth.com/specifications/bluetooth-core-specification/ (accessed May 09, 2020).

[28] “Insight Into ESP32 Sleep Modes & Their Power Consumption,” *Last Minute Engineers*, Dec. 23, 2018. https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/ (accessed Feb. 01, 2020).

[29] “Oscilloscope Preamplifier LNA10,” *AlphaLab, Inc.* https://www.alphalabinc.com/product/lna10/ (accessed May 11, 2020).

[30] “A Guide to Understanding LiPo Batteries,” *Roger’s Hobby Center*. https://rogershobbycenter.com/lipoguide (accessed May 10, 2020).

[31] “Battery Discharge Methods – Battery University.” https://batteryuniversity.com/learn/article/discharge_methods (accessed May 10, 2020).

[32] “CP2102-9.pdf.” Accessed: May 11, 2020. [Online]. Available: https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf.

[33] “Voltage Regulator - CMOS, Low Noise, Low-Dropout 1.pdf.” Accessed: May 13, 2020. [Online]. Available: https://www.onsemi.com/pub/Collateral/NCP500-D.PDF.

[34] “NCP703-D.pdf.” Accessed: May 13, 2020. [Online]. Available: https://www.onsemi.com/pub/Collateral/NCP703-D.PDF.

CODE APPENDICES

A: Final Sensor and Gateway Node Firmware Files

https://github.com/jawahl/Thesis_Firmware/tree/master/FINAL_FIRMWARE_FILES

B: Python Firebase Scraper Script

https://github.com/jawahl/Thesis_Firmware/tree/master/FINAL_FIRMWARE_FILES/firebaseScrape.py