Software Defined Radio Bluetooth Low Energy Communication System with ThingSpeak Integration

Fivos Kavassalis*, Faith Kurtz*, Jonathan Lee*
*Worcester Polytechnic Institute, Worcester, MA 01609 USA
Email: fikavassalis@wpi.edu, fkurtz@wpi.edu, jlee4@wpi.edu

***Abstract*—When creating an end-to-end radio system, packet error rate (PER) is vital in determining the viability of that radio system. This analysis is also commonly applied onto Software Defined Radio (SDR) system designs like the one that we try to recreate. Our project uses two ADALM-PLUTOs to create an end-to-end software defined radio system that uses the Bluetooth Low Energy (BLE) standard. Our design uses BLE to transmit and receive data through related data channels and determines the system's PER. The decoded data is uploaded to ThingSpeak and statistics are stored in the cloud for later analysis.**

***Keywords*—**
Bluetooth Low Energy, Software Defined Radio, Packet Error Rate, ADALM-Pluto

I. Introduction

Bluetooth Low Energy (BLE) is a wireless personal area network technology that allows the transmission of a suitable amount of data to devices, such as smartphones, that are high on energy consumption and the interaction of mobile devices with generic sensors measuring physical properties to provide users an improved customer experience that "senses" their cyber-physical environment. The big advantage of the BLE technology is its high degree of energy efficiency. Consequently, it is cost saving and has better functioning performance for mobile devices and that it can be connected to a large number and variety of mobile devices and operating systems, thus allowing interaction on multiple platforms and cross-device use cases.

Although BLE is part of the same specification as Bluetooth, in reality it is not treated as the same technology since BLE is not reverse-compatible with Bluetooth [1]. BLE, as a new technology, is mostly centered around IoT applications where small amounts of data need to be transmitted at lower speeds. It is designed as a short-range transmission technology, thus it can be used to send data 2-5 m in reality, although this distance can be extended up to 30 m albeit with higher energy consumption. In addition, the BLE's upper limit in terms of its data rate is 2 Mbps although in practice a lower rate is expected.

Another motivation that we had for this project was that our project could become the prototype or proof of concept to a system that was designed to use a multitude of SDRs to send information to a main information hub that could record and store all that information for later processing. One way that our system could be used is helping monitor moisture levels within fertile lands or irrigated farms. By having a system such as this constantly collecting data from those farms, they could be watered whenever it was necessary and help to optimize the water usage in those farms and help the surrounding areas conserve water. This task would be especially important and helpful within communities where water is scarce and an essential commodity.

In this paper, we detail the process of creating a system that collects data as strings and converts that information into bits that are coded, modulated and manipulated to create a transmission signal to send over BLE to another BLE receiver that should be within range of a wireless router so that the data could be decoded, demodulated, and reformed into bits that can be converted back into strings of information for uploading into a central hub where that information can be stored and used for a variety of purposes. This system is designed to be especially general so that it can be used for more than just the previously mentioned situation. Since this design is made to allow various transmitters to send data that will be read and uploaded to the cloud, it could also use it to pick up other messages or data that the user can preset the system to read themselves. This design will also allow the system to have flexibility in its usages and make full use of its wide area data collection network properties.

II. Background

A BLE device can be divided into three sections: the application, the host, and the controller. The focus of this project is primarily on the controller section, which contains the Physical layer and Link layer of the communication stack.

A. Physical Layer Specifications

BLE uses the 2.4GHz to 2.4835GHz band [2]. This band is evenly split into 40 channels, 3 of which are used for advertising and the rest of which are used for data [2]. Each channel has a bandwidth of 2MHz [3]. The band is divided into these channels so that frequency hopping spread spectrum (FHSS) can be used, which decreases narrowband interference. This project will be implemented on the ADALM-Pluto, which supports a frequency range of 325 MHz to 3.8 GHz and a channel bandwidth of up to 20MHz [4].

BLE uses Gaussian Frequency Shift Keying (GFSK) with a modulation index of 0.5 [2]. BLE version 5 has the option of either 1Mbit/s or 2Mbit/s [3].

A BLE transmitter must have a transmit power of at least 0.01 mW (-20 dBm), but no more than 100 mW (20 dBm) [3]. The receiver must have a minimum sensitivity of -70 dBm for a BER of 0.1% [3]. These requirements are within the capabilities of the Pluto SDR.

B. Link Layer Specifications

Each functioning BLE device must have a Bluetooth device address, a unique 48 bit number assigned to a Bluetooth device which differentiates it from other devices [2]. There are two types of these addresses: public device addresses , which are fixed and registered with the IEEE Registration Authority, and random device addresses which are not [2].

When BLE devices are not connected to each other, they operate in advertiser and scanner roles. An advertiser device sends advertiser packets, and a scanner device scans for these packets [2]. The packets can be used to discover potential slave devices or to broadcast small amounts of data for applications which do not require an established connection [2]. Each advertising packet includes header information (including the Bluetooth device address of the sender) and 31 bytes of advertising data [2]. Since the advertiser and scanner are not synchronized, advertising packets are only received by a scanner when they happen to be on the same channel [2].

Once the BLE devices establish a connection, they assume master and slave roles. A master device initiates and manages connections with slave devices [2]. A slave device follows the master device's timing [2]. When a connection is created between a master and slave device, the master device sends a connection request packet which includes the frequency hop increment used for FHSS and 3 connection parameters: connection interval, slave latency, and connection supervision timeout [2].When BLE devices are connected, they use data packets to communicate (rather than advertising packets). The usable data payload of a data packet is 216 bits, though this number is often reduced as additional protocols are added to the communication system [2].

When operating on the BLE standard, each received packet is checked against a 24-bit CRC regardless of the role of the BLE device (master, slave, etc.) [2]. When there is a failure, the receiving device will request a retransmission [2]. An interesting feature of this system is that there is no limit to retransmissions -- the cycle repeats until the packet is sent successfully [2].

Figure 1 below shows a summary of the establishment and use of a connection to send a single packet between two BLE devices. This is borrowed from [3].

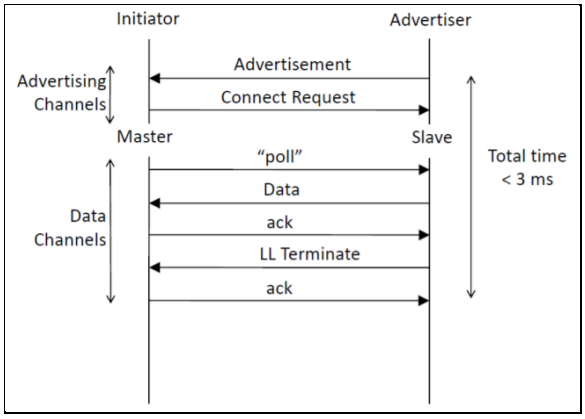

Figure 1: Overview of BLE Communication System

III. Final Proposed Design

To create a complete BLE system from info source to info sink, we will be using two ADALM-Plutos, two computers for Matlab computations and configuration, and a microphone. As seen in figure 2, we first receive information in the form of a string from the microphone. This string is processed by our Matlab program to produce a signal for the transmitter to transmit. Once the signal is generated and the transmitter is configured, the signal is sent

to the ADALM-Pluto to begin transmitting. After travelling through the channel, which is about a foot of air in our case, the signal is recorded by our receiver. The recorded transmission is then processed in order to determine the information payload's data bits for converting into usable information. After we determine the usable information, whether it be "hello world" or a floating point number, we upload that information up to ThingSpeak and store it within the database.

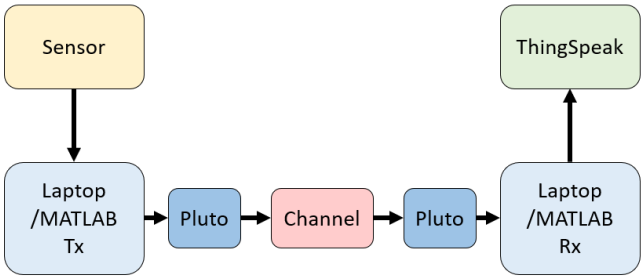

Figure 2: Block Diagram of the Overall Design's Data Transmission Path

IV. Methodology

Our BLE Implementation can be split in four sections: the sensor interface, the transmitter, the receiver and the ThingSpeak integration. In this part of our report we are going to analytically explain how we executed each section.

A. Sensor Interface

We incorporated a sensor in our system in order to transmit messages with input data from that sensor. More specifically, we connected the LM393 microphone to the ESP32-PICO-KIT microcontroller. We were receiving the microphone's sensor data in 100 ms intervals using the microcontroller's analog pin. Each message that we eventually wanted to transmit consisted of 10 microphone readings. This message was written as a string to the serial port periodically, with the baud rate set to 9600 bps, once 10 new microphone samples were stored. Furthermore, the ESP32 microcontroller has a 12-bit ADC so the values that we expected to read from the microphone were in the range of 0 to 4095. However, our LM393 microphone has a potentiometer from which we could alter its sensitivity. Since it was set to low sensitivity, the values that we were receiving were closer to 0 than to 4095 and therefore the sound excitation had to be larger in order for the readings to change substantially. Figures 3 and 4 below display the environment, figure 5 shows the flowchart of our sensor interface. Figure 6 displays the messages written from the microcontroller to the serial port.

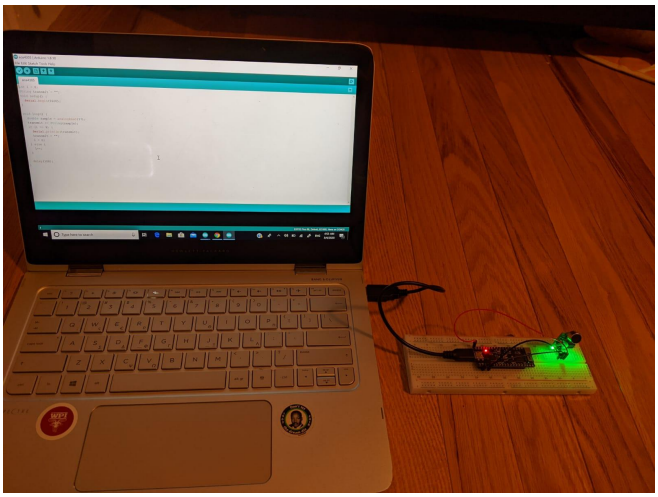

Figure 3: Top Level Environment of the Sensor Interface

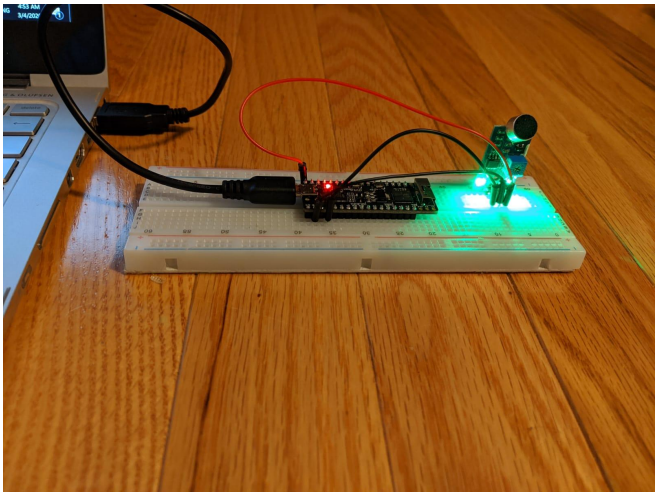

Figure 4: Connection of LM393 Microphone with ESP32-PICO-KIT Microcontroller

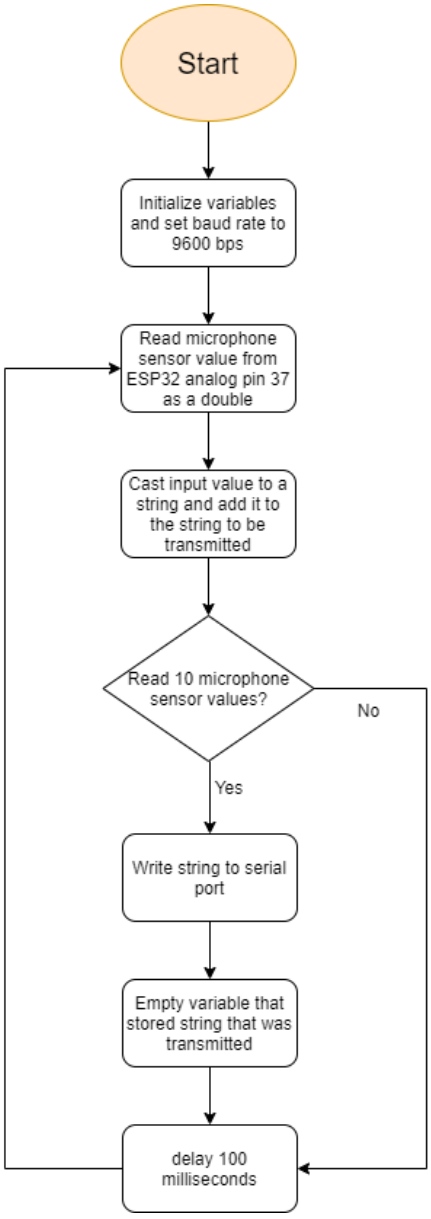

Figure 5: Flowchart Displaying the Functionality of Microcontroller

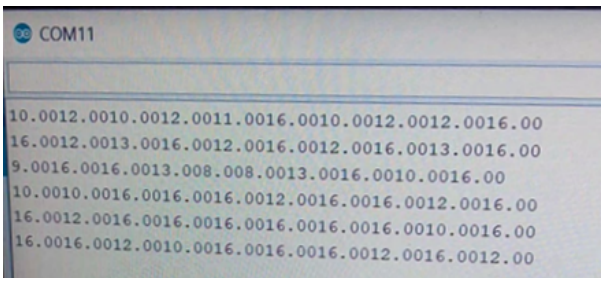

Figure 6: Messages Written to Serial Port by Microcontroller

B. Transmitter

The transmitter SDR's functionality was implemented in Matlab. We performed repetitive transmission of messages by implementing an infinite loop. We initially checked whether the 'Communications Toolbox Library for the Bluetooth Protocol' is installed. If not, an error is thrown. If it is installed, we configured and generated an advertising channel protocol data unit (PDU). Then, our program listened to the serial port, where a string of fifty characters is transmitted since each microphone reading is of type double and therefore has two digits after the decimal point. Once we stored the 10 microphone sensor values as a string to our Matlab program, we converted the string to hex and we reshaped it into a one dimensional array. Afterwards, we transmitted the message over BLE in that form through the data channel by generating the data channel PDU and inserting that array as an input argument. Subsequently, we initialized the parameters required for generating the baseband two IQ waveforms for the advertising and the data channel message bits respectively. We set the physical layer mode to uncoded 1 Mbps (LE1M), samples per symbol to 8, channel index to 37 and access address length to 32. Furthermore, we defined the access address value in hex and in binary and we set the symbol rate to 1e6. Once all of the initializations were done, we generated the baseband IQ waveform for the advertising channel and the data channel message bits (one for each channel). Then, we initialized the parameters required for the signal source by setting the center frequency to 2.402 GHz, the frame lengths to their corresponding IQ waveform lengths, the number of frames to 1e5 and the transmitter front end sample rate to 8 Msps. We set the signal source to 'ADALM-PLUTO' and we checked whether the hardware support package existed. If it did not, then an error was thrown. Otherwise, the PLUTO that was connected to the computer was discovered, we initialized it as a transmitter system object and we transferred the baseband data to it. Finally, we released the signal sink objects after transferring both IQ waveforms and the program is started all over again. A flowchart of the transmitter is shown in figure 7.

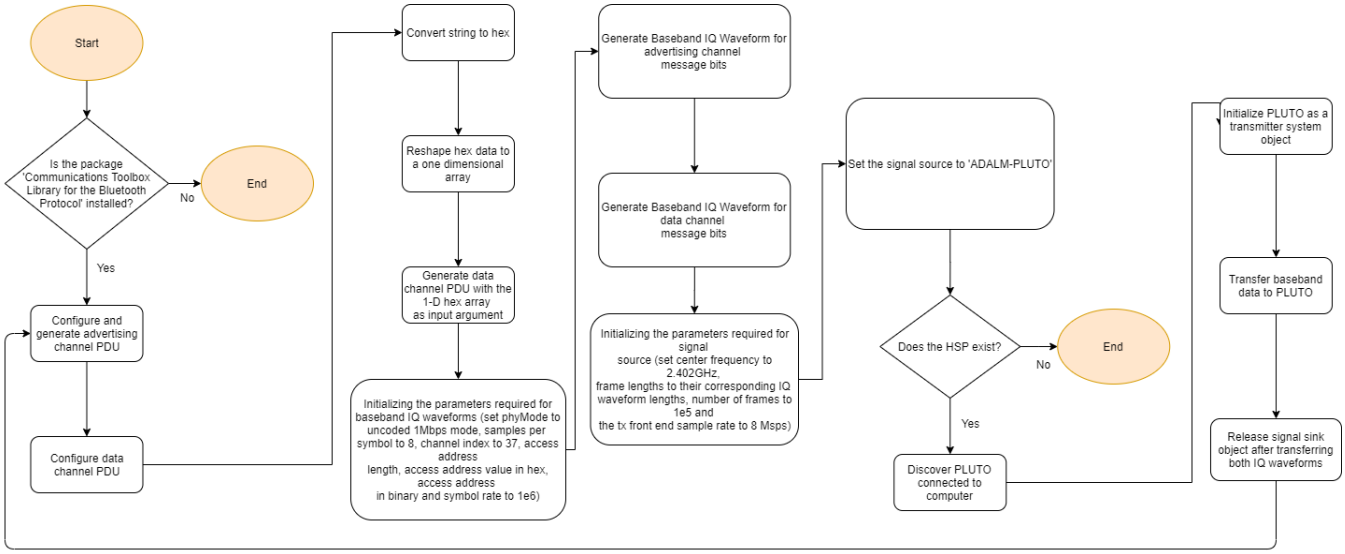

Figure 7: Transmitter Flowchart

C. Receiver

In order to receive the signal that we are sending from the transmitter SDR, we needed to configure another SDR for receiving bluetooth transmission, create the algorithms to pull out the bits from the encoded signal, and convert those bits into the information that we are trying to recieve. A flowchart that outlines the entire receiver process is shown in figure 8 below.

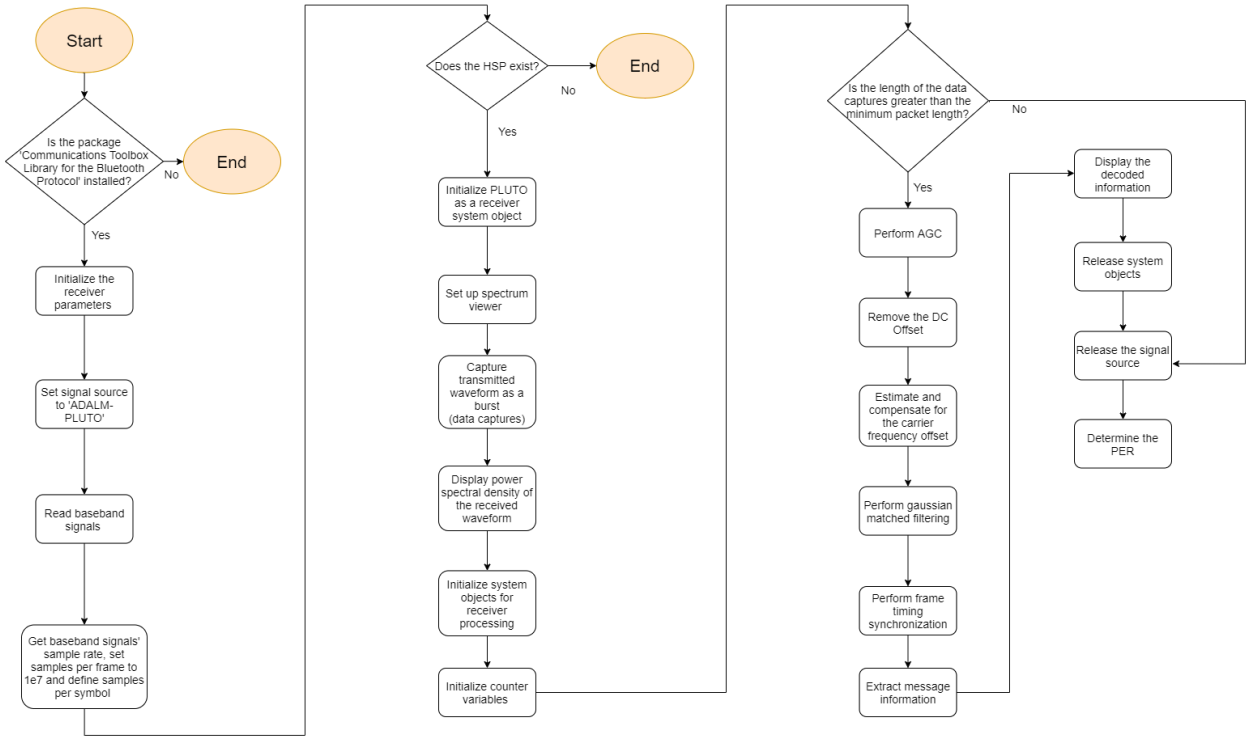

Figure 8: Receiver Flowchart

Step 1 - Pluto Receiver:
Within the ADALM-Pluto SDR, we needed to configure the receiver antenna to sense the incoming signal with the parameters within table 1.

Sampling Frequency:	1 MegaBits per Second
Center Frequency:	2.402 GigaHertz
Baseband Sample Rate:	8 MegaSamples per Second
Samples per Frame:	100 KiloSamples per Frame

Table 1: Receiver Configuration

Within Matlab, this can be done by altering the data within sdrrx's data structure.

Step 2 - Automatic Gain Control:
Automatic gain control(AGC) simply regulates the signal to have a maximum power or magnitude. In our situation, we were regulating the signal to have a maximum power of 2. We also reduce the amount of gain that the signal can receive so that the signal does not start clipping and distorting. To run AGC, we use the comm.AGC function provided by Mathworks to create an object that will apply AGC to the signal.

Step 3 - Remove DC Offset:
After receiving the signal amplitudes, there is a DC offset due to the fact that we cannot receive a "negative" signal. Thus, we convert the signal from amplitudes from 0 to 2 into amplitudes from -1 to 1.

Step 4 - Frequency Compensation:
Now we have a signal that can be synchronized using the comm.CoarseFrequencyCompensator function from the communications toolbox. With this premade frequency compensator provided by Mathworks, we will be able to remove any frequency offsets that we may have picked up during transmission.

Step 5 - Gaussian Pulse matching:
After correcting the signal for frequency offsets, we need to perform Gaussian Pulse matching. We perform this task by convoluting the compensated signal with "bleparam.h" from the BLE configuration data structure and return the central part of the convolution which should be the length of the originally entered signal.

Step 6 - Timing Synchronization:
Within our function, we perform timing synchronization at the same time as we search for preambles. This is executed using the comm.PreambleDetector function provided by Mathworks' Communications toolbox. Once we find preambles, we use those "detections" to synchronize time within our custom helper function.

Step 7 - GMSK Demodulation:
Within the helper function as well, we demodulate the data by using the ble.internal.gmskdemod function which will demodulate the applied signal using the Gaussian minimum shift keying demodulation with the preset parameters from the BLE configuration data structure.

Step 8 - DeWhitening:
Also within the Helper function that we made is a process to dewhiten the data. Both the transmitter and receiver code use the same internal Matlab function for whitening and dewhitening. The Matlab "whiten" function whitens or dewhitens a binary input using specified 127-bit whitening sequence, circular array shifting, and the generator polynomial x^7+x^4+1.

Step 9 - CRC Check:
Within the Helper Function we also pull out the exact bits that we need to manipulate into usable information. We extracted those bits by using the bleLLDataChannelPDUDecode function by Mathworks to first decode the information into bits and then output the data payload into the "LLpayload" output.

Step 10 - PER:
Once we have extracted the data bits, we can compare the original data bits with the extracted ones in order to determine our packet error rate which is the ratio of incorrect packets of total packets sent. When testing, we received a packet error rate that ranged from seven to fifteen percent error rate.

D. ThingSpeak Integration

In order to upload the data to ThingSpeak, we first had to convert it into an appropriate format. The data bit recovery function outputs a hexadecimal string. We used various Matlab functions and loops to convert that data into a column vector of decimal measurements which could then be uploaded to ThingSpeak.

Data can be easily uploaded to ThingSpeak using Matlab's *thingSpeakWrite* function. A channel ID and write key can be obtained when the channel is created on the ThingSpeak website.

V. Experimental Results

In this section, we will discuss our experimental results as well as the challenges we encountered.

A. Results

In our final implementation, we were transmitting floating point values taken from our microphone. Figures 9, 10, and 11 show the received spectra at different points

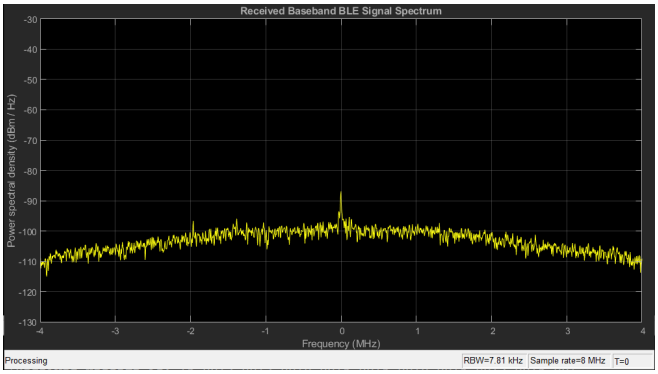

Figure 9: Received Spectrum When No Packets Are Being Transmitted

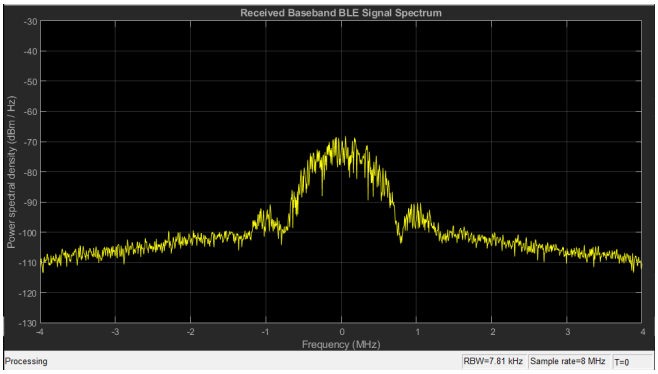

Figure 10: Received Spectrum When Packets Are Being Transmitted, Calculated PER = 0.125523

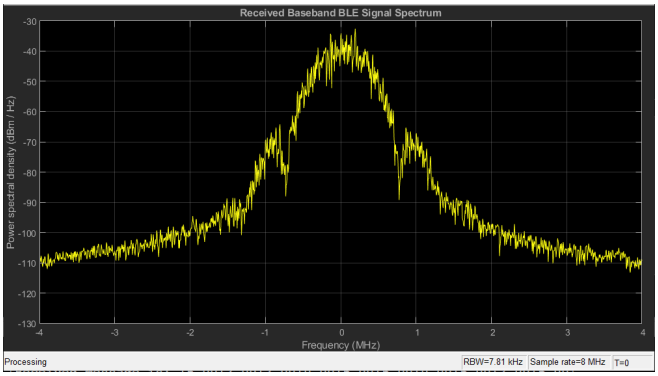

Figure 11: Received Spectrum When Packets Are Being Transmitted, Calculated PER = 0.098522.

As shown in figures 10 and 11 above, the spectrum of the received signal varies with time. Both of these spectra were obtained for the same payload value being transmitted: '16.0012.0012.0010.0016.0016.0010.0016.0012.0016.00' For further discussion of the observed PER values, see section VI.

We successfully decoded the transmitted message at the receiver whenever a packet was detected. Our receiver code in Matlab displayed the decoded message in the command window as follows:

'Received Message is: 16.0012.0012.00...'

We did not notice any occurrences of incorrectly decoded bits or symbols, only of total packet loss most likely due to timing issues.

Finally, after all data has been recovered and formatted, it was uploaded to ThingSpeak. Figure 12 shows the ThingSpeak graph of data. The graph includes 80 data points, with 10 data points uploaded every 15 minutes.

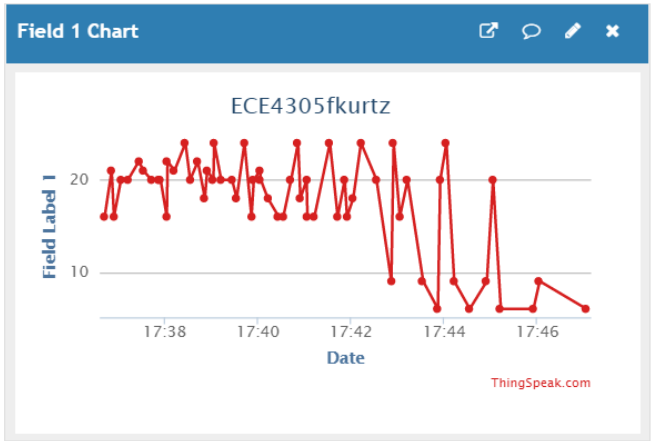

Figure 12: ThingSpeak Data Graph

B. Issues and Challenges

Originally, the provided helper function for recovering the Physical Bits from the BLE transmission was meant for only pulling out the physical bits for an advertising channel since the decoder that we implemented within it was only for a BLE advertising channel. Thus, our group needed to look through the original helper function and replace this decoder with a data channel decoder and alter some of the peripheral code to accommodate this change.

One of the more confusing changes that we needed to implement was setting the CRC to the same CRC that the transmitter was using. This issue was just a problem of understanding the sample code, but with time and Kuldeep, we figured out what to do.

Another issue that we had was that the data was no longer being stored in the same way that we thought. Originally, the Advertising data decoder sent everything to the "cfgLLAdv" output so we thought that it would work the same with the data channel decoder, but we were wrong since the new decoder required another output to store the bits that were extracted. Additionally, that output needs to be initialised or else nothing will be stored.

VI. Discussion

When working on this project, our group also had to consider time constraints and as a result, was left with a few areas that we knew we could improve upon. These are the areas that we believe that we could improve: transmission packets design, receiving packets without a CRC, and packet error rate.

A. Transmission Packet Design

During transmission, we are sending 10 values that we read from our sensor. We could improve this by finding a way to send those 10 bits one by one instead of 10 at a time. We believe that the issue we were getting when trying to transmit in this setup was due to the fact that Matlab could not keep up with the processes that needed to occur to send the data as previously specified. To fix this issue, we would need to use a different program that would be able to handle these processes in real time or we would need to continue debugging to find out if there was some form of hardware issue that we missed.

B. Receiving Packets Without a CRC

Another part of our project that we could improve is how we were receiving data. Sometimes an error where the array lengths did not match occurred after transmission. We believe that this is due to white noise producing a signal similar to our preamble. However, the following data that the system processes would not have a CRC resulting in the decoder not removing the CRC bits at the end and the array lengths being mismatched after the decoder.

C. Packet Error Rate

We initially thought that the variation in the packet error rate was due to different data measurements being transmitted. However, to test this theory, we transmitted the same data in multiple "identical" experiments, and obtained different PER values each time. For this reason, we believe the PER variation was due to timing variation when running the system on Matlab, and also potentially due to outside interference since the BLE band is so commonly used.

VII. Conclusion

Through this project, we were able to learn how to implement a BLE data channel and send data that can be reconverted back into characters or floating point numbers for storage within ThingSpeak. We enjoyed this project and have learned about BLE in depth as we were trying to implement it. We were also surprised at how poor Matlab is at trying to keep up with real-time systems and how difficult it is to implement an entire standard that we use on a nearly daily basis. Overall, this was a fun and interesting project with many sources of information to learn from and we enjoyed the development and learning process we followed when trying to generate this system.

References

[1] Honkanen, M., Lappetelainen, A., Kivekas, K. et al (2004) Low end extension for Bluetooth 2004 IEEE Radio and Wireless Conference 19-22 September 2014 IEEE pp 199-202.

[2] K. Townsend et al, Getting Started with Bluetooth Low Energy. 2014.

[3] Silicon Labs, "UG103.14: Bluetooth®LE Fundamentals," [Online]. Available: https://www.silabs.com/documents/public/user-guides/ug103-14-fundamentals-ble.pdf. [Accessed: Jan. 30, 2020].

[4] Analog Devices, "RF Agile Transceiver" AD9363 datasheet, 2016.