University of Southern Indiana
Pott College of Science, Engineering, and Education
Engineering Department

8600 University Boulevard
Evansville, Indiana 47712

Electroencephalogram Controlled Electric Wheelchair

Nicole Matthews & Andrew McClain
ENGR 491 – Senior Design
Fall 2021

Approved by: ____
Faculty Advisor: Arthur Chlebowski, Ph.D. Date

Approved by: ____
Department Chair: Paul Kuban, Ph.D. Date

ACKNOWLEDGEMENTS

In this section, we would like to acknowledge Dr. Arthur Chlebowski for advising us through this process and our families for all their support and encouragement.

ABSTRACT

The purpose of this project was to create an alternative method to control an electric wheelchair using an electroencephalogram (EEG). The primary goal of this project was for an EEG to collect data and transmit it wirelessly via Bluetooth to a microcontroller on board the wheelchair. This data is then processed and used to control the motor functions of the wheelchair.

The EEG headset used in this project was the Unicorn Hybrid Black. This headset has eight data electrodes as well as two reference electrodes. Multiple microcontrollers were analyzed to determine the best fit for this project with the nRF52840 chip on the PCA10056 development kit ultimately being selected. Matlab and Simulink were used to receive and process the signal from the EEG headset. Then the logic to create the signal for the wheelchair motors and emergency brake controls was designed and loaded to the microcontroller onboard the wheelchair.

The final system uses the EEG headset to collect data that is processed through a computer and outputs a signal to an Arduino that is connected to one nRF52 microcontroller which then transmits that signal via Bluetooth to the nRF52 microcontroller onboard the wheelchair.

TABLE OF CONTENTS

TABLE OF FIGURES

<u>ELECTROENCEPHALOGRAM CONTROLLED ELECTRIC WHEELCHAIR</u>

1 INTRODUCTION

According to the National Spinal Cord Injury Statistical Center, there are currently approximately 294,000 people in the United States with spinal cord injuries. That number is increasing daily with about 18,000 new spinal cord injuries every year. Of these spinal cord injuries, 12.3% of these injuries result in complete quadriplegia/tetraplegia, and the next 47.2% resulting in incomplete quadriplegia/tetraplegia, see Figure 1. With incomplete quadriplegia, some movement of the arms or legs may be possible, but it is limited. [1] Because of this, a large portion of those with spinal cord injuries may need to use alternative control methods for an electric wheelchair. These alternative control methods can give those suffering with this type of injury a sense of independence that may otherwise not be possible.

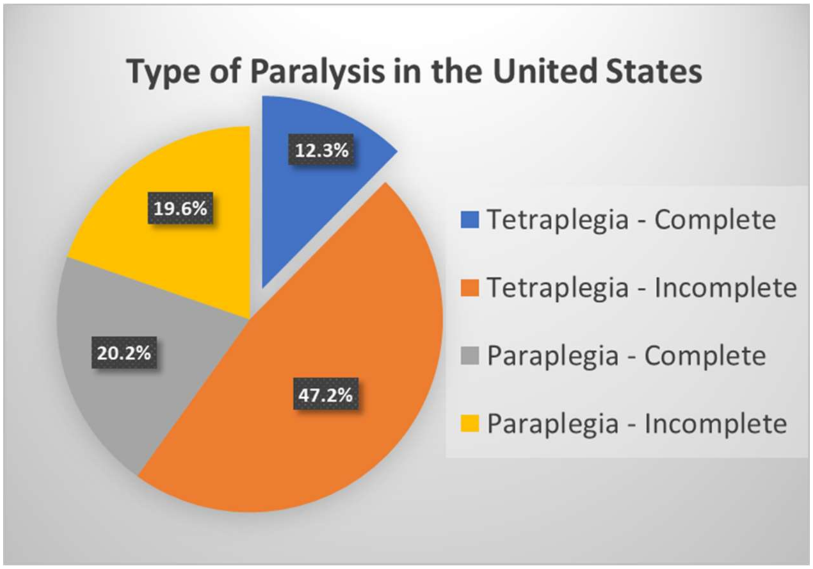

Figure 1: Types of Paralysis in United Stated

This project was done to help those who may not have as much freedom of movement as they would like. Increasing mobility options for people without many options can increase a person's quality of life significantly. This quality-of-life boost can lead to an overall increase in a person's welfare. People with quadriplegia or multiple amputations face very limited mobility options. Electric wheelchairs are most commonly driven with a joystick to be controlled with the hand, see Figure 2. When that is not possible due to lack of motor function in the hands, the options are more limited. This project gives another method of controlling a wheelchair, using brain signals.

Figure 2: Joystick Controller

An electroencephalogram (EEG) was used to collect brain signals and drive a wheelchair. With the idea in mind that the user has very limited motor functions, every function was designed to be controlled via the EEG headset. This includes turning the system ON and OFF with a series of eye blinks and driving the motors based on the very specific input from select electrodes.

2 PROJECT BACKGROUND

This project is the continuation of another design team's project. That team used the Unicorn Hybrid Black headset to transmit data to a laptop computer that was running MatLab Simulink. The data was processed there, and the signal to drive the motors was connected serially to an Arduino Uno which sent a signal to the motor drivers powering the motors. [2]

The steps taken in the current project aim make the system wireless. The system designed in this project would ideally only include the headset and one microcontroller to collect the data and output the appropriate drive signals.

3 SYSTEM DESIGN

The main system can be broken down into two main systems as shown in Figure 3. The EEG headset collects information from brain waves and transmits the information wirelessly. The wheelchair subsystem receives and analyzes the transmission then initiates the wheelchair movement by powering the correct motors in the appropriate direction.

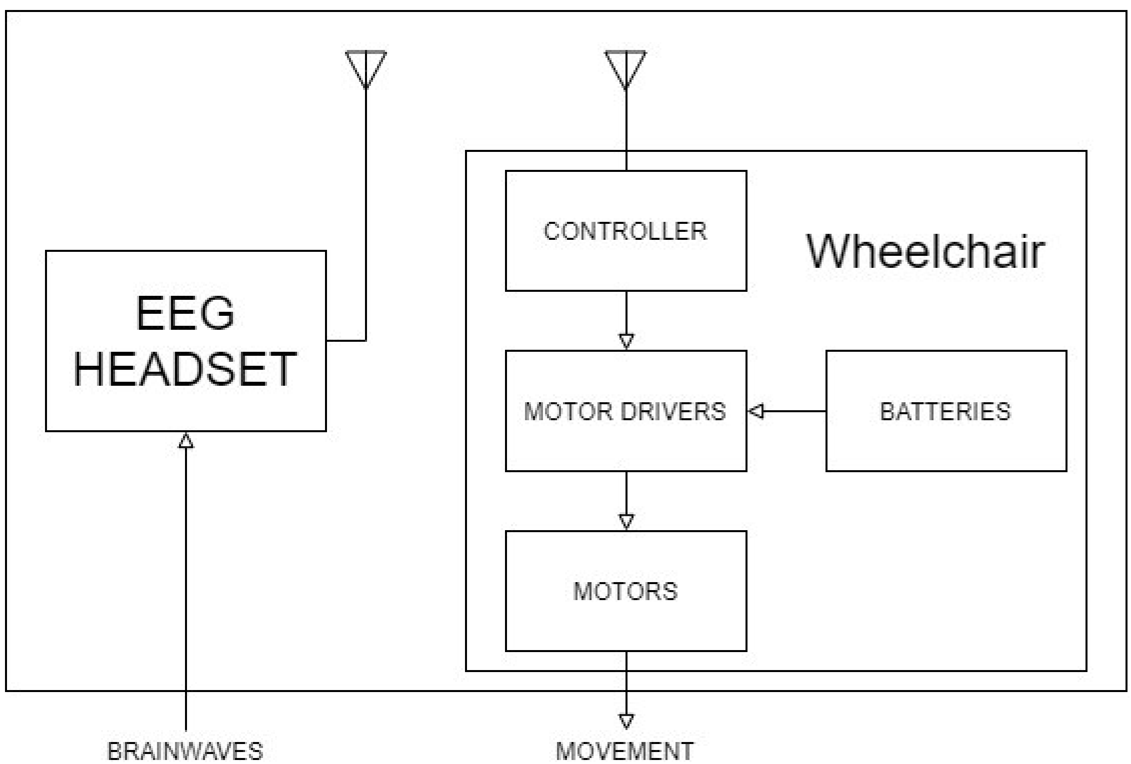

Figure 3: Low Level System Architecture.

Designing the system to operate in the manner described above was a very complex and could more easily be accomplished by first breaking those systems into smaller subsystems, which could be changed in the future to reach the end goal of a fully embedded system. This modified block diagram can be seen in Figure 4. The system has been broken down into three subsystems called A, B, and C.

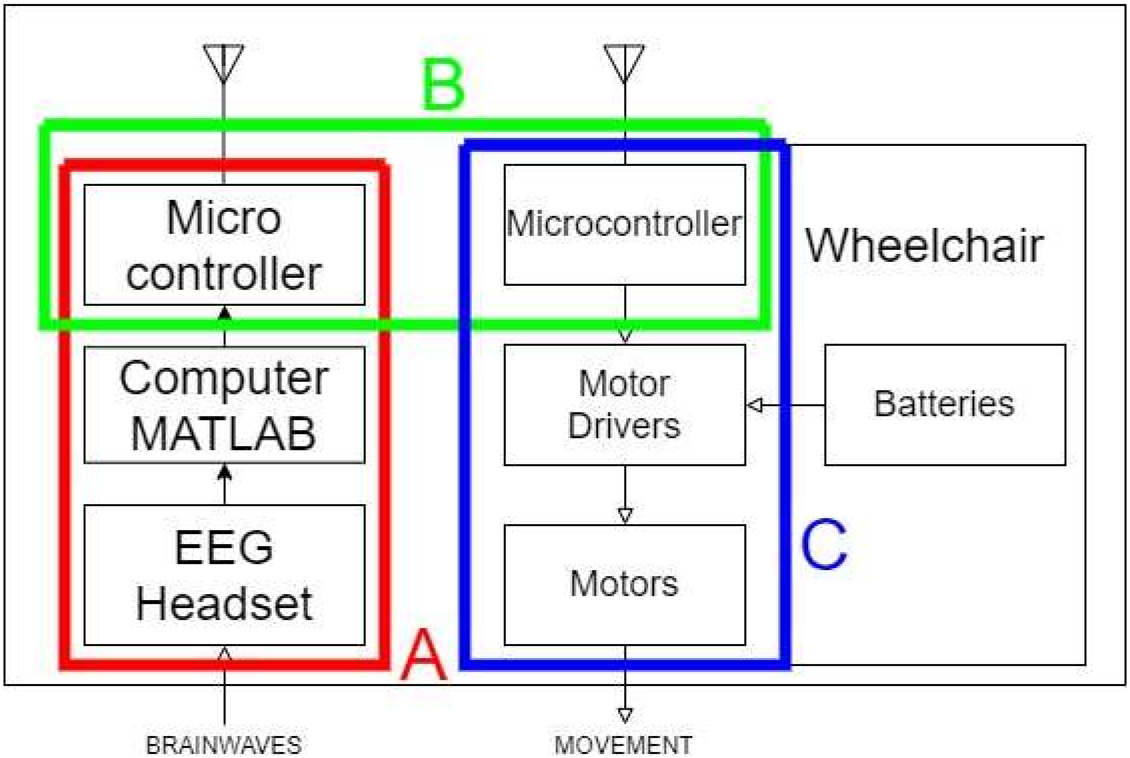

Figure 4: Modified block diagram to illustrate the subsystems designed individually.

With this modified system, the first subsystem is section A, where the EEG signal was processed on a computer separate from the microcontroller on board the wheelchair. This signal is then passed to an Arduino which outputs the appropriate wheelchair operation signal. The output pin of the Arduino is physically connected to the button of a microcontroller that is located off the wheelchair.

The second subsystem was the microcontroller-to-microcontroller Bluetooth connection labeled section B. In this system, the microcontroller located off the wheelchair would connect to the microcontroller on the wheelchair and send the drive information which was then processed and used to turn on the appropriate motors on the wheelchair. This is discussed in more detail in 5.2.8.

The final subsystem is the control of the wheelchair and its motor functions from the microcontroller on board the chair denoted as section C. The microcontroller needs to supply a pulse width modulated signal to drive the motors as well as a direction to drive the motors in. Beyond that, the microcontroller needs to turn the emergency brake on and off when the wheelchair is ready to be driven.

4 EEG HEADSET

Several EEG headsets are commercially available. Many have similar features to the Unicorn Hybrid Black and none were found to have significant enough advantages to justify purchasing a different headset for this project. According to the Unicorn Hybrid Black User Manual, the Unicorn Hybrid Black headset has eight data electrodes as well as two reference electrodes. When the headset receives a start acquisition command, it begins to transmit a 45-byte signal with the battery life, electrode data, accelerometer, gyroscope, and counter data. This signal is transmitted at a frequency of 250 Hz. [3] A profile view of the headset being worn can be seen in Figure 5. This data collected by the headset was used to control the operations of the motors on the wheelchair.

Figure 5: Profile View of Unicorn Hybrid Black Headset

4.1 TESTING

Testing needed to be done to determine how the data would be output from the headset. A timed trial was the next step to see if there were identifiable responses to stimuli. Together these tests provide a good data set to analyze.

4.1.1 Initial Testing

Following instructions provided on the Unicorn website the headset was connected to the Unicorn Suite software. Several initial tests were run to determine what type of information is provided by the software. During a test the headset was turned on and different options within the software were utilized. Pressing the play button displays a signal from the EEG. An example of this can be seen in Figure 6. There is also a visual display for the electrodes that turn from yellow to green when there is a good signal. It was noted in the information material that it can take up to a couple minutes for the signals to normalize. [3] When pressing the record button, the same information is available on the display and when the recording is stopped a comma-separated-value (.csv) file is saved with data for each electrode as well as data for the gyroscope, accelerometer, and counter.

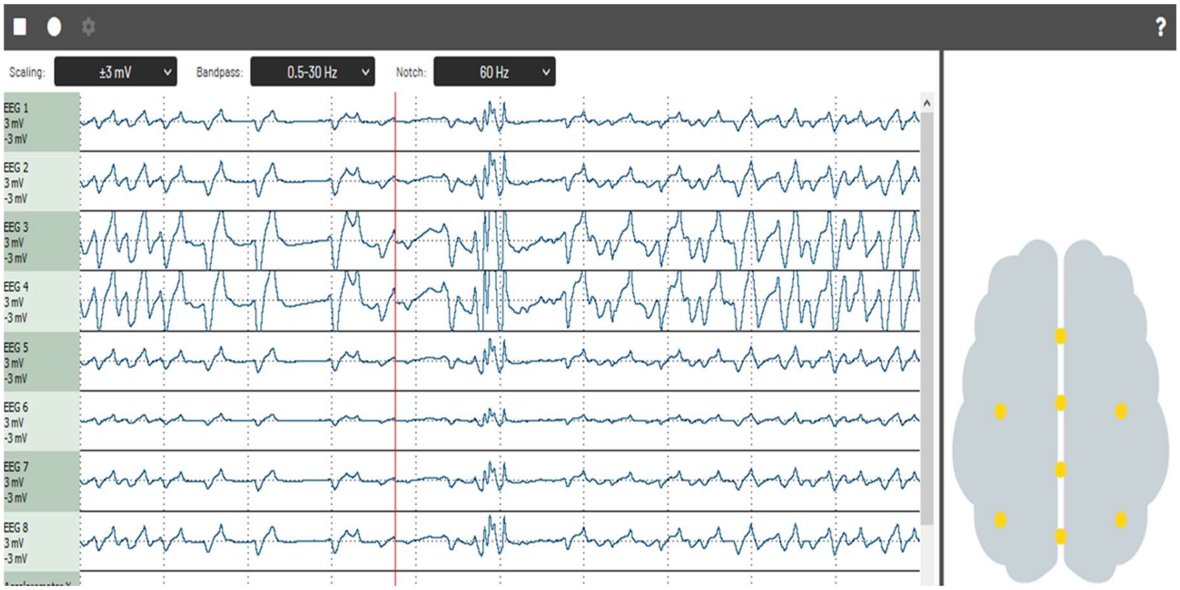

Figure 6: Unicorn Suite EEG Sample.

4.1.2 First Trials

To help with determining how to process the data and determine what to use for triggering moving and stopping events, an initial series of tests was completed. Data was collected from one subject. The test procedure was discussed with the subject prior to beginning the test. The subject was asked to relax during the initial stages of the test to allow the electrodes to get a good signal and to provide a baseline. The subject was sitting with a relaxed posture, hands hanging at sides and feet resting on the crossbar of a stool. For the first series of tests the subject kept his eyes open and stared at a fixed point. The subject was given several tasks and asked to relax between tasks. The tasks were described to the subject beforehand to allow for questions and to give the subject time to think of needed responses. The tasks were as follows: curl left toes, curl right toes, squeeze left index finger and thumb together, squeeze right index finger and thumb together, clench teeth, think about your favorite food, think about eating something sour, think about a time you were scared, think go left, think go right, think go straight, and think stop. The subject chose a relaxing thought and returned to that thought between each task. This entire test was repeated with the only change being the subject kept their eyes closed for the duration of the test.

4.2 ANALYSIS

4.2.1 Excel

Initially the data in the .csv file format was viewed using Microsoft Excel. A sample of this work data can be seen in Figure 7. Some basic graphs of specific sections of data were created to visualize any potential responses to the stimuli in the initial trial. An example graph can be seen in Figure 8. Each color in the graph represents data from a single electrode. The data begins with the stimulus prompt being given, halfway through a relax prompt is given. Different electrodes had higher values at different points but comparing results of several different stimuli there were no consistent high points that could be used for control. With over 48000 rows of data, it proved very difficult to scroll through and select specific data of interest, so the data was loaded into Matrix Laboratory (MatLab) for further analysis.

EEG 1	EEG 2	EEG 3	EEG 4	EEG 5	EEG 6	EEG 7	EEG 8	Accelerometer X	Accelerometer Y	Accelerometer Z	Gyroscope X	Gyroscope Y	Gyroscope Z	Battery Level	Counter	Validation
34987.84	37400.89	-20529	-18152.7	41822.7	41989.75	22919.67	41345.49	-0.019	1.006	-0.01	0	0	0	73.333	1	1
138059.9	150779.7	-111642	-115377	163443.9	192327.2	28125.71	172705.8	-0.016	1.012	-0.017	0	0	0	73.333	2	1
172212.4	197329.9	-245756	-335303	174748.4	341803.9	-192100	261723.4	-0.014	1.014	-0.018	0	0	0	73.333	3	1
14996.48	26586.29	-260775	-577549	-151156	285678.6	-573500	138729.7	-0.018	1.014	-0.02	0	0	0	73.333	4	1
-79068	-128166	-75669.5	-612547	-560576	117665.3	-557463	-54783.2	-0.017	1.013	-0.022	0	0	0	73.333	5	1
147519.5	26272.32	169717.7	-325547	-594018	92350.04	13167.84	-73752.1	-0.017	1.01	-0.023	6.104	-4.822	-3.784	73.333	6	1
480586.8	353977.8	309142.7	101530.7	-239407	216866	575461.3	116220.3	-0.019	1.009	-0.025	3.204	1.373	-1.556	73.333	7	1
614776.4	563697.1	315337.7	375782.2	180445	361146.6	750851.9	400358	-0.02	1.01	-0.025	2.747	2.838	-1.19	73.333	8	1
558956.4	581527.8	257523.4	406622.7	458596.1	472692.8	654773.7	592946.1	-0.017	1.009	-0.025	2.563	3.387	-1.007	73.333	9	1
452352	487289.1	219996.6	324006.7	561085.2	516645.8	495191.7	578342.5	-0.016	1.007	-0.025	2.533	3.51	-0.702	73.333	10	1
386836.8	397628.2	236059.2	281394.4	532251.7	480364.4	435075.6	462733.2	-0.011	1.004	-0.027	2.625	3.479	-0.519	73.333	11	1
413816.8	405452.7	260729.6	333176.2	500887.9	445632.8	523590.5	418136.5	-0.01	1.006	-0.031	2.808	3.448	-0.336	73.333	12	1
476004.9	469968.3	241331.8	414546.2	539689.5	462287.2	594080.8	456288.3	-0.012	1.007	-0.029	2.899	3.387	-0.153	73.333	13	1
447688	459054.3	200770.6	422469.9	542108.1	443968.9	487527.9	465351.5	-0.01	1.008	-0.031	2.991	3.357	-0.092	73.333	14	1
341164.4	360619.7	195689.1	364428.5	437700.8	328736.9	313262.3	384189.4	-0.011	1.007	-0.031	3.113	3.265	-0.061	73.333	15	1
281696.1	288986.9	217937.8	336875.9	333054.9	198161.7	253665.7	263146.6	-0.01	1.007	-0.032	3.174	3.143	-0.183	73.333	16	1
267433.9	265395.7	213037	357149.3	295078.2	129189.3	249659.8	172198.1	-0.008	1.004	-0.031	3.235	3.113	-0.214	73.333	17	1
179305.7	188985.9	178828.6	356849.5	231135.3	74437.6	138765.9	107554.1	-0.008	1.005	-0.03	3.418	3.204	-0.183	73.333	18	1
22770.62	45116.84	164891.4	318779	95511.79	-11242	-18879	33253.96	-0.009	1.006	-0.031	3.54	3.204	-0.397	73.333	19	1
-56049.8	-42611.2	180672.3	295668.8	5712.163	-68928.9	-41473.9	-30751.3	-0.01	1.009	-0.034	3.693	3.082	-0.458	73.333	20	1

Figure 7: Comma Separated Value File View in Excel for 20 Data Points

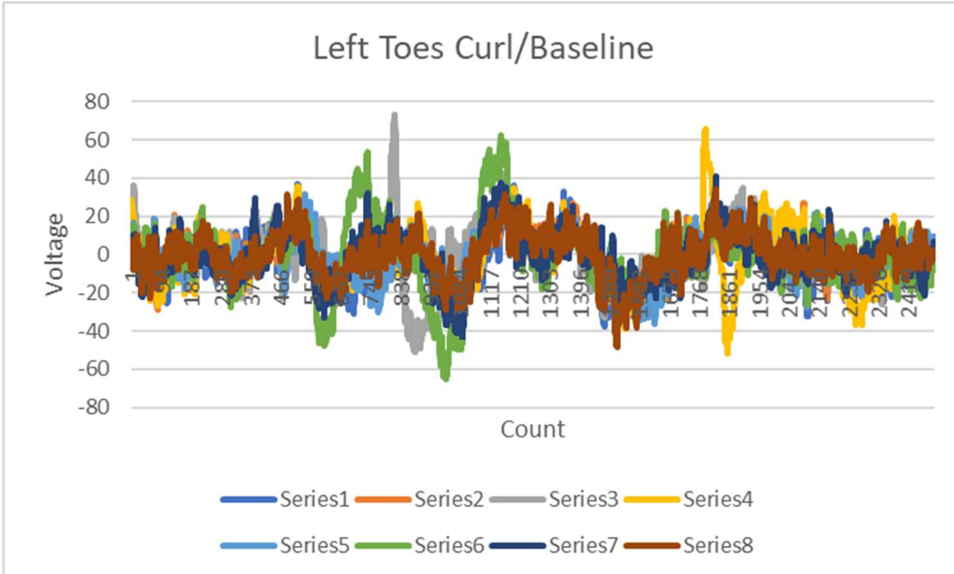

Figure 8: Excel Graph of One Stimulus Response and Baseline Period

4.2.2 MATLAB

MatLab is an excellent program for manipulating large arrays of data. It was possible to upload the data from the .csv file and graph the electrode frequencies over specific periods of time to see if there were any noticeable reactions to stimuli. While the responses to different stimuli did vary from one another, the differences were not enough to be able to write a program to recognize a certain event as a trigger for wheelchair movement. It was noted that physical movement caused more noise in EEG channels than thinking about different things did.

The next idea was to look at which area of the brain was most active as a result of the different stimuli instead of looking at changes in frequency of each electrode individually. To accomplish this the distance between electrodes was measured in two-dimensional space with electrode 3 being assumed to be at the origin point. The locations of the other electrodes in reference to the origin were measured in inches. The resulting values can be seen in Figure 9. The graph shows the approximate placement of each electrode. Electrodes 1, 3, 5 and 7 run along the midline of the skull from the top of the head to the base of the skull. Electrodes 2 and 6 are on the left side of the head while electrodes 4 and 8 are on the right side. As can be seen in the photos of a person wearing the headset in Figure 10, all the electrodes are located closer to the rear portion of the head. The locations were added to the data in MatLab. The frequency amplitude data for each electrode was multiplied by the x and y locations for the corresponding electrode. The data from all eight of the electrodes multiplied with the x-locations was averaged to calculate a single x-location that would have the highest frequency and the same was done with the data multiplied by the y-locations. This resulted in a single x-y coordinate for the set of data captured by one point in time. The plot function was used in MatLab to plot the coordinates over 5 second sections of time to see if any distinct areas were stimulated based on some of the given stimuli. Figure 11 shows several of these plots with the movement left or right depicted on the x-axis and the movement front to back depicted along the y-axis. It is noted that there was more movement front to back than left to right but overall, no specific area of activation was seen for any one stimulus. In order, the stimuli depicted in these plots are rest, left toes curl, rest, right toes curl, rest. Ideally, some of the non-rest stimuli would produce a dense area somewhere other than the center of the graph, but the densest area for each sample would still be near the center of the of the region. At this point the decision was made to utilize a program called EEGLab that works with MatLab and offers advanced filtering and visualization options for interpreting EEGs.

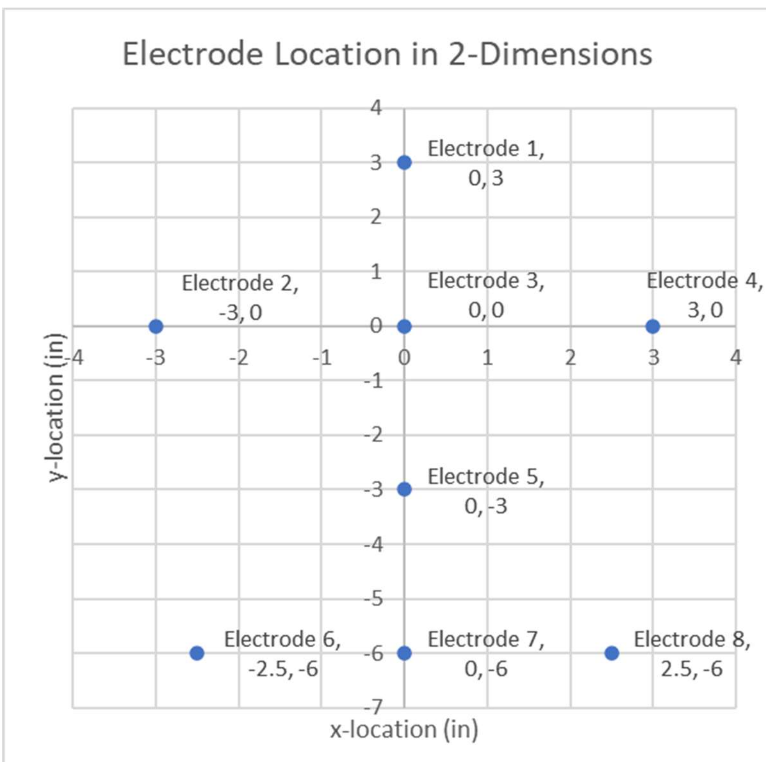

Figure 9: Electrode Locations in 2-Dimensions.

Figure 10: Photos of Headset Cap with Electrode Location (Profile and Front Views).

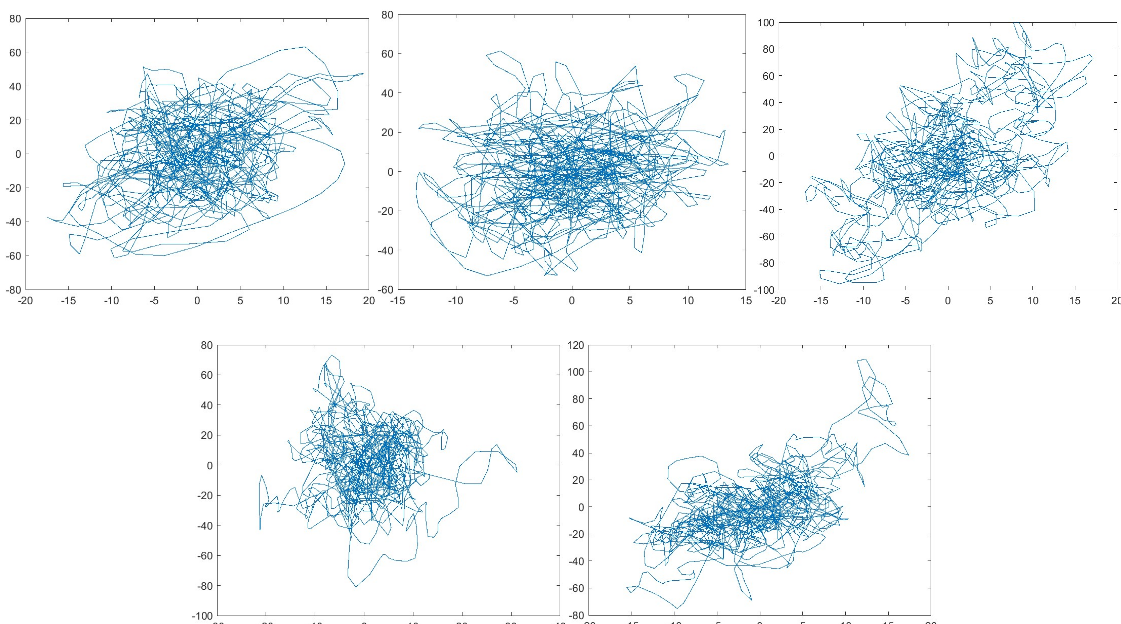

Figure 11: Heat Map for 5 Second Intervals.

4.2.3 EEGLab

EEGlab provides tutorials to guide users though the initial setup and basic use of the product. The recommended steps from The EEGLAB Wiki Tutorial were followed as described below.

Installing EEGLAB – The program was downloaded and run in MatLab as described. The program uses a command in MatLab to open a graphical user interface (GUI). The GUI can be used to accomplish the additional steps.

Quickstart – This step was only used to familiarize the user with some of the basic functionality of the program such as accessing datasets and scrolling through the data plot.

Dataset Management - This step covered how to save modify and delete datasets which becomes important when using the preprocessing steps.

Import Data - There were several types of data that needed to be imported to view the data. The continuous data was contained in the .csv file that was uploaded to MatLab. The data needed to be transposed for the program to correctly recognize the eight channels. Event data was also imported to show when stimuli were introduced and when rest periods occurred. Standard locations of a 10-20 electrode configuration, the configuration used for the Unicorn Hybrid Black were available with EEGLab. The included electrodes were selected based off data provided by unicorn and the three-dimensional location information for each electrode was added.

Preprocess data - This step included filtering, re-referencing and resampling the data. The data was filtered as recommended, but did not need to be re-referenced as the Unicorn Hybrid Black has two reference electrodes. Resampling was not used as the sampling rate was already in the desired range.

Reject Artifacts - This step is used to find and get rid of bad channels and data. Through the analysis it was found that Channel 6 had much more noise than any of the other channels and would likely need removed if the data was to be used. There were no distinct sections of bad data identified by the researchers.

Extract Data Epochs - Data Epochs are used to look specific data based on time of event. For the purpose of this test looking at data from 2-3 seconds after a stimulus seemed to be a reasonable window to determine if there was any identifiable response to the stimulus.

Plot Data - This was the last step completed by the researchers in the project. Data was viewed as frequency over time through the scroll options and as a power spectrum. [4]

This program includes many more advanced options than were able to be explored as a part of this project. This program could prove very useful in the future if many more trials were completed with the same stimulus repeated numerous times to find trends. The program also has options for studies for multiple subjects. With the constraints of this project, it was determined that a distinct, recognizable, and repeatable method of controlling the wheelchair would not be easy to identify and transfer to a microcontroller.

4.2.4 Methods to Control Using EEG signals

Additional research was done to see how other people were using EEG signals to control functions. Some commonalities were identified. Most of the projects utilized EEG equipment with included functions for attentiveness or something similar and used these functions are triggers to control the projects. [5] One project used a Neurosky chip that "amplifies and pre-processes the incoming neural data, outputting real-time estimated levels of attention, of relaxation, and frequency band power, using custom algorithms." [6] Another researcher used a Cyton Biosensing Board. [7]

Some projects utilized eye-blinks as a good solid indicator for control. [8] The placement of electrodes on the Unicorn Hybrid Black is not ideal for picking up eye-blinks, so three electrodes were removed from the cap and placed near the eyes, one on each temple and one in the center of the forehead.

Many projects also utilized a laptop or smartphone for receiving, processing and/or transmitting their data. [5] [8] [9] A previous senior design team used the Simulink add-on for the Unicorn to collect and filter data from the headset. [2] The decision was made to investigate using Simulink to receive, process and transmit data for this project.

4.2.5 Simulink

The senior design team previously working on this project used the Simulink add-on for the Unicorn Hybrid Black and an Arduino UNO to control the motion of the wheels. [2] The decision was made to build off that work. Figure 12 shows the Simulink setup using a 60Hz notch filter and a Bandpass filter from 15 to 30Hz. The notch filter helps to eliminate noise from the electrical system. Using a Bandpass filter in this range helps to isolate Beta waves that occur when a person is alert or active. Using a moving average helps to eliminate false positives that might be caused by movement or noise but averaging the EEG data that included positive and negative values basically removed all the artifacts of interest. To eliminate this problem and increase the amplitude of the artifacts, the data was squared before putting it through a moving average. A function was added to look at the three electrodes of interest. Figure 13 show how the raw data appears. The yellow box is the response from a single hard blink with both eyes. This large amplitude was consistent for single blinks. After several trials it was determined that a single electrode in the center of the forehead could produce a consistent artifact when a hard blink occurred.

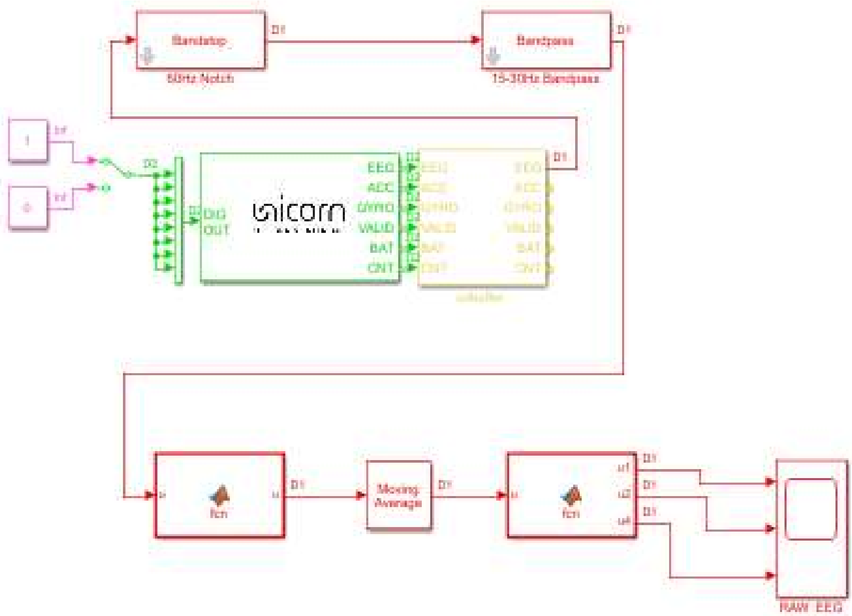

Figure 12: Simulink Raw Data**.**

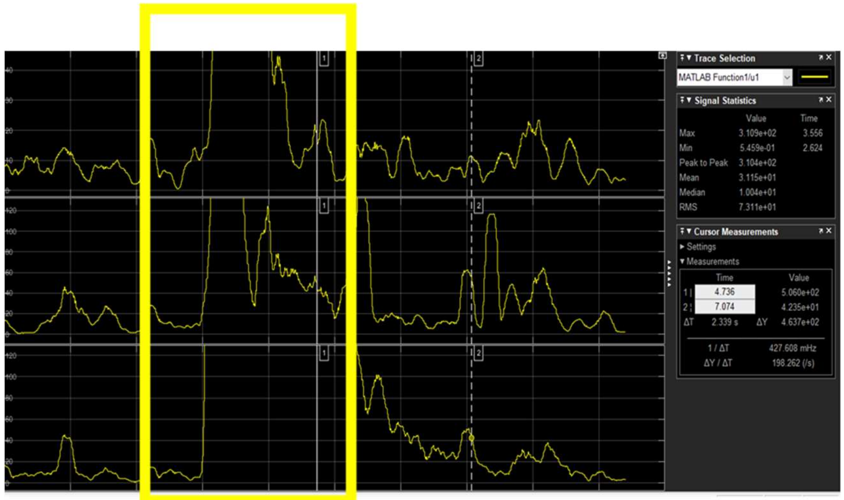

Figure 13: Simulink Raw Data Output.

The next test was to see if the filtered signal could be used as a means of control. The Arduino UNO was connected to Simulink. A function was written to send a high signal if the electrode amplitude was above a threshold level and a low otherwise. A single LED on a breadboard was connected to a PIN on the Arduino. A digital output was used to send to signal through the Arduino lighting the LED if a high signal was received. The modified Simulink setup can be seen in Figure 14. When the program was run, the LED lit up immediately. This was to be expected as the EEG signals start out large before they have time to normalize. After a short time, the LED went off. With a hard blink the LED lit up again for a short time. The LED setup with the LED lit up can be seen in Figure 15. When viewing the output of the control function the blinks were regularly identified but movement was not. This indicates that this should be a good way to control a function. A difficulty with using the Arduino is that the digital output does not occur in real time. This creates larger and larger lag between when the blink occurs and when the LED lights up. Several other output types including serial and servo were considered as potential alternatives to the digital output. None of the alternatives worked to create a close to real time output of the control information.

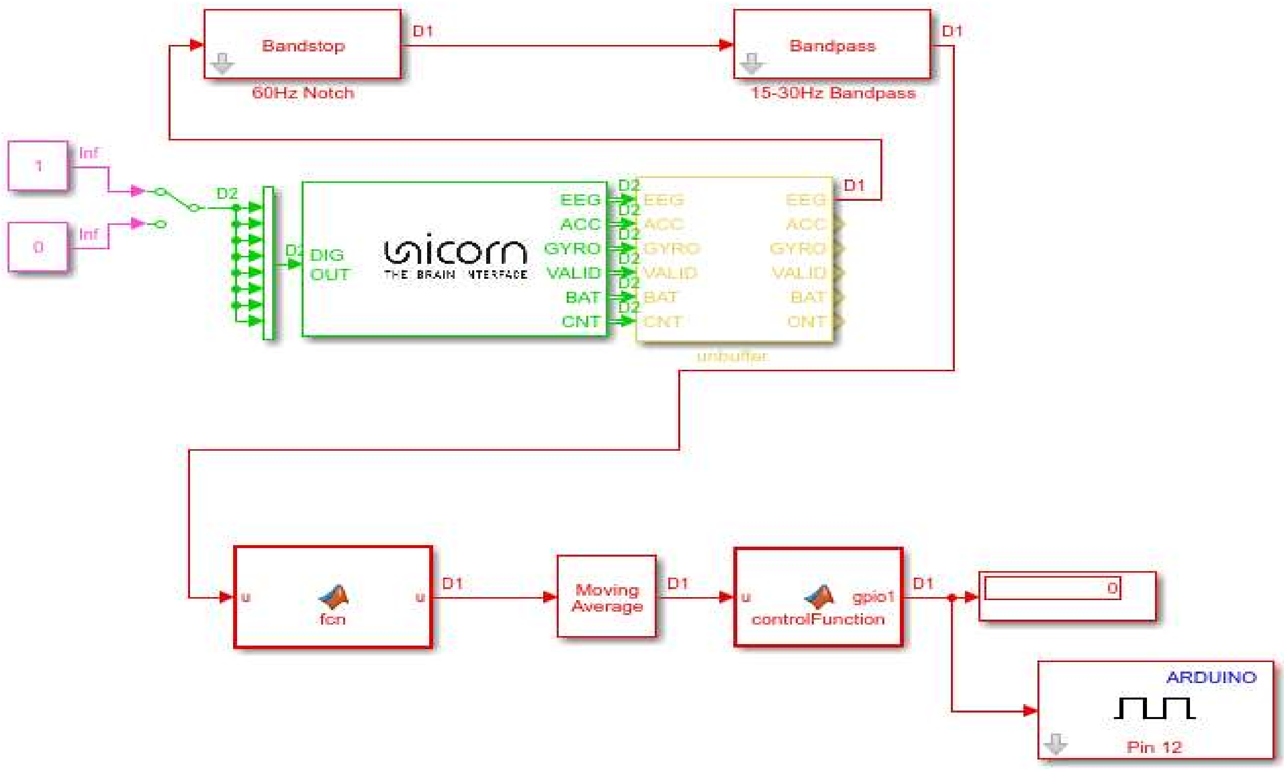

Figure 14: Simulink Arduino Output

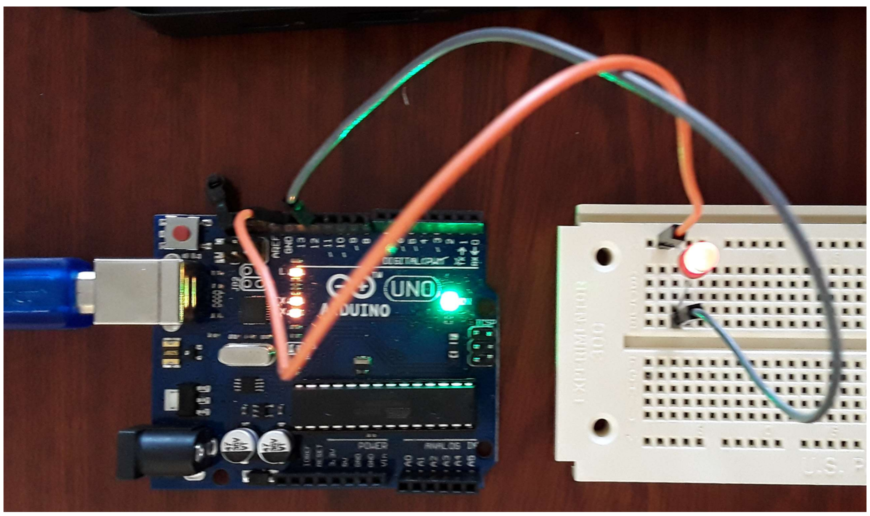

Figure 15: Arduino Output for LED Control

4.3 *MatLab From Simulink for Control*

The earlier work processing data in MabLab revealed that there was very little lag even processing large amounts of data while working in MatLab. This led researchers to consider the possibility of sending the data from Simulink to MatLab for writing the control function and sending the data to the Arduino. There is a simout block in Simulink that can be used for this purpose. As can be seen in Figure 16, the simout block was used to replace the control function and the Arduino Digital Out that were used in Figure 14. The Arduino was initiated in MatLab and a script written to accomplish the steps in Figure 17. A detailed flow chart of the logic for the script can be seen in Figure 18 and the code can be found in Appendix E. The code starts the simulation and waits for 2 seconds while Simulink collects data then pauses the simulation so that data becomes available in MatLab. That data is then averaged over a short amount of time and compared to an experimentally developed threshold to determine if there was significant enough activity to turn on the light, or eventually the motor. If the threshold is met or exceeded the light or motor turn on, if not the light or motor is turned off. Then more data is processed. When all data has been processed the simulation is restarted for another two seconds to collect more data and the process is repeated. This allowed for much closer to real time response and kept the system from getting so behind that it could no longer process. Additional testing could help determine optimal numbers of data points to average and how long to collect data to get even closer to real time results.

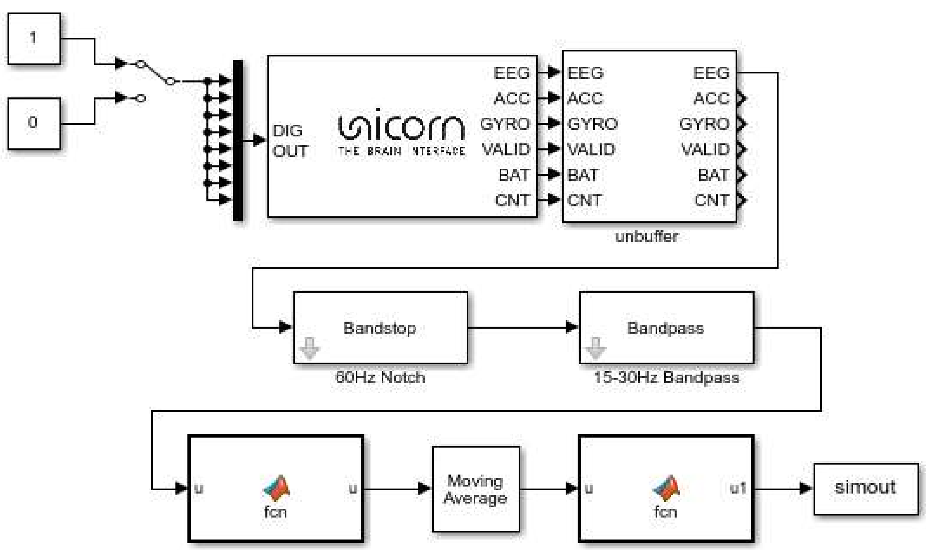

Figure 16: Simulink Data Out to MatLab

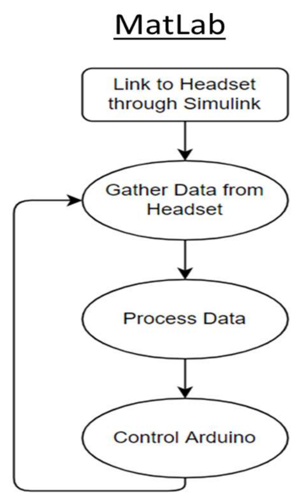

Figure 17: Data processing and Arduino Control through MatLab

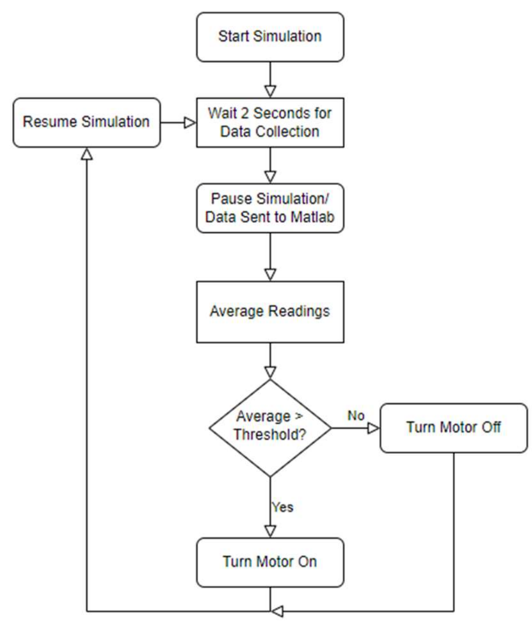

Figure 18: Flow Chart for Collecting and Processing Data in MatLab

4.4 OUTPUT TO MICROCOMPUTER

Once an LED was consistently lit with a blink as shown in Figure 15, the wire connecting the Arduino the LED was instead used to connect the Arduino to a button being pressed on the Bluetooth transmitter to move the wheelchair forward.

5 MICROCOMPUTER

5.1 Code Structure

The code to operate the microcontrollers consists of two subsystems. The first being the information that will be used to physically control the motor functions of the wheelchair, and the second being the Bluetooth connection between the two boards. Each of these subsystems are built using functions to initialize the different components of the systems. All these functions are then initialized in the main program and runs in an idle state while waiting for interrupts to start execution. The main code can be seen in Appendix C and Appendix D.

The signal being received on the microcontroller onboard the wheelchair is an 8-bit word, that is used to control the four motor functions of the wheelchair. Using conditional statements, one command will operate the corresponding motors when necessary.

The Bluetooth connection of the two microcontrollers is an adaptation of the Blinky example that is provided in the software development kit provided by Nordic Semiconductor. This program is designed to connect a designated peripheral microcontroller to a specific central microcontroller based on the connection parameters (primarily the name of the devices) being met.

5.1.1 Acceleration Control

Smoother transitions between stop and full speed need to be incorporated into the system in future iterations. The current setup of the system takes the wheelchair from a stop position directly to full speed without any ramp up of speed. This can lead to jarring and injury of passengers. This is a very important aspect of future design iterations.

5.1.2 Pulse Width Modulation

The first iteration of motor controls was done on the nRF51 microcontroller. This board did not have a preprogrammed pulse width modulated (PWM) signal peripheral. Because a PWM signal was required to drive the motors, this signal had to be created using hardware shorts and the timers on the microcontroller. The PWM signal was created by turning the output of a pin on and off at a rate of 1kHz.

The timer and the general-purpose input/output tasks and events (GPIOTE) peripherals were used to create the PWM signal. The timer runs on an 8MHz clock and has an event at 8000 clock cycles, which is at 1ms. The calculation for the speed of the signal can be seen in Equation 1. This is the basis for the 1kHz signal.

$$Signal = \frac{Clock\ Speed}{Capture/Compare\ Event}$$

Equation 1: Calculating the speed of the PWM signal.

Setting the duty cycle of the PWM signal is a matter of when to turn the signal to the off position within each cycle of the PWM signal. This was accomplished using a capture/compare register to designate when that signal is set to zero. For a 50% duty cycle, the signal would be turned off at 4000 clock cycles. This signal can be seen measured on an oscilloscope in Figure 19.

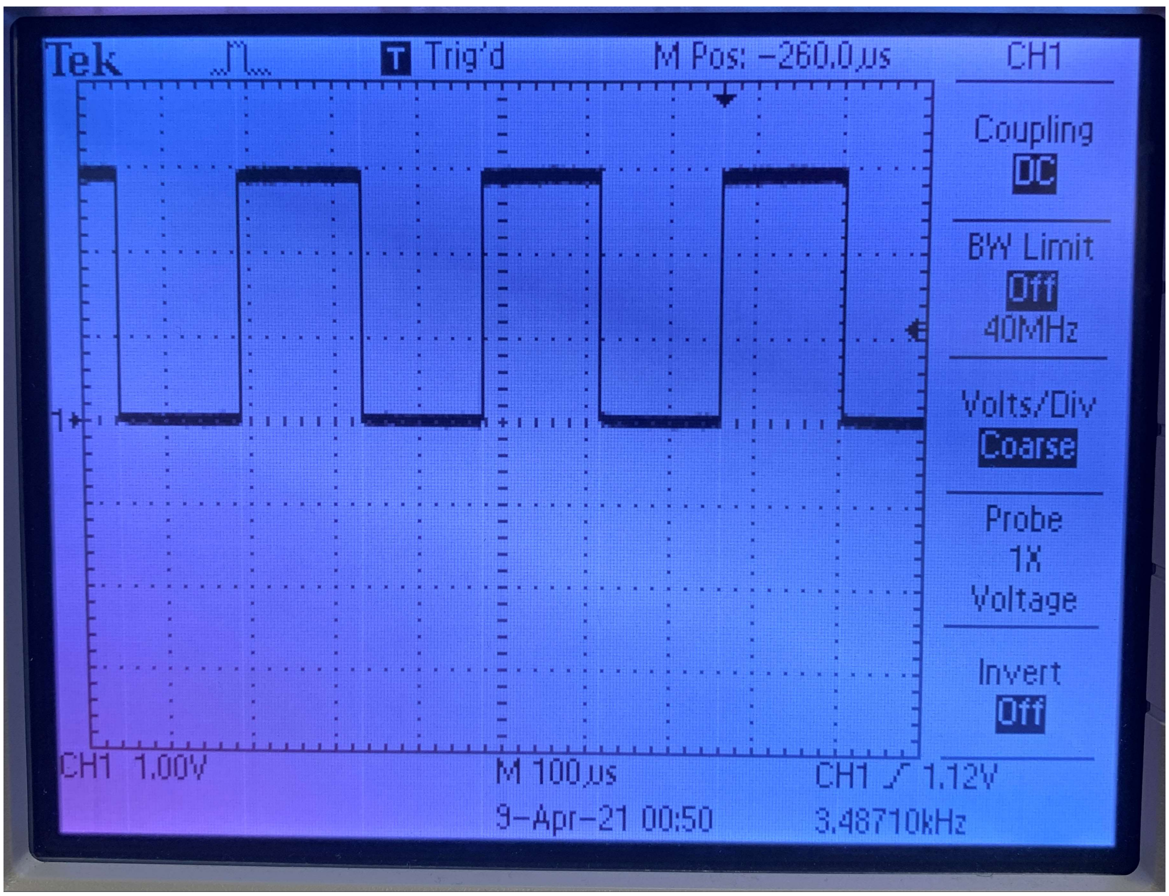

Figure 19: PWM signal generated on nRF51 microcontroller.

To correctly turn the motors off, the duty cycle of the PWM signal had to be set to zero. This is done by changing the output of the pin to zero for all time. This was accomplished by checking for a stop signal while the PWM signal was on a low cycle. This had to be checked in this manner because if it stopped the timer on a high cycle rather than the low cycle, the duty cycle would be set to 100% instead of 0% and the wheelchair would drive out of control. This can be seen in Figure 20 in the timer interrupts. If the stop command is not given, the wheelchair will resume driving in the same direction that it was driving.

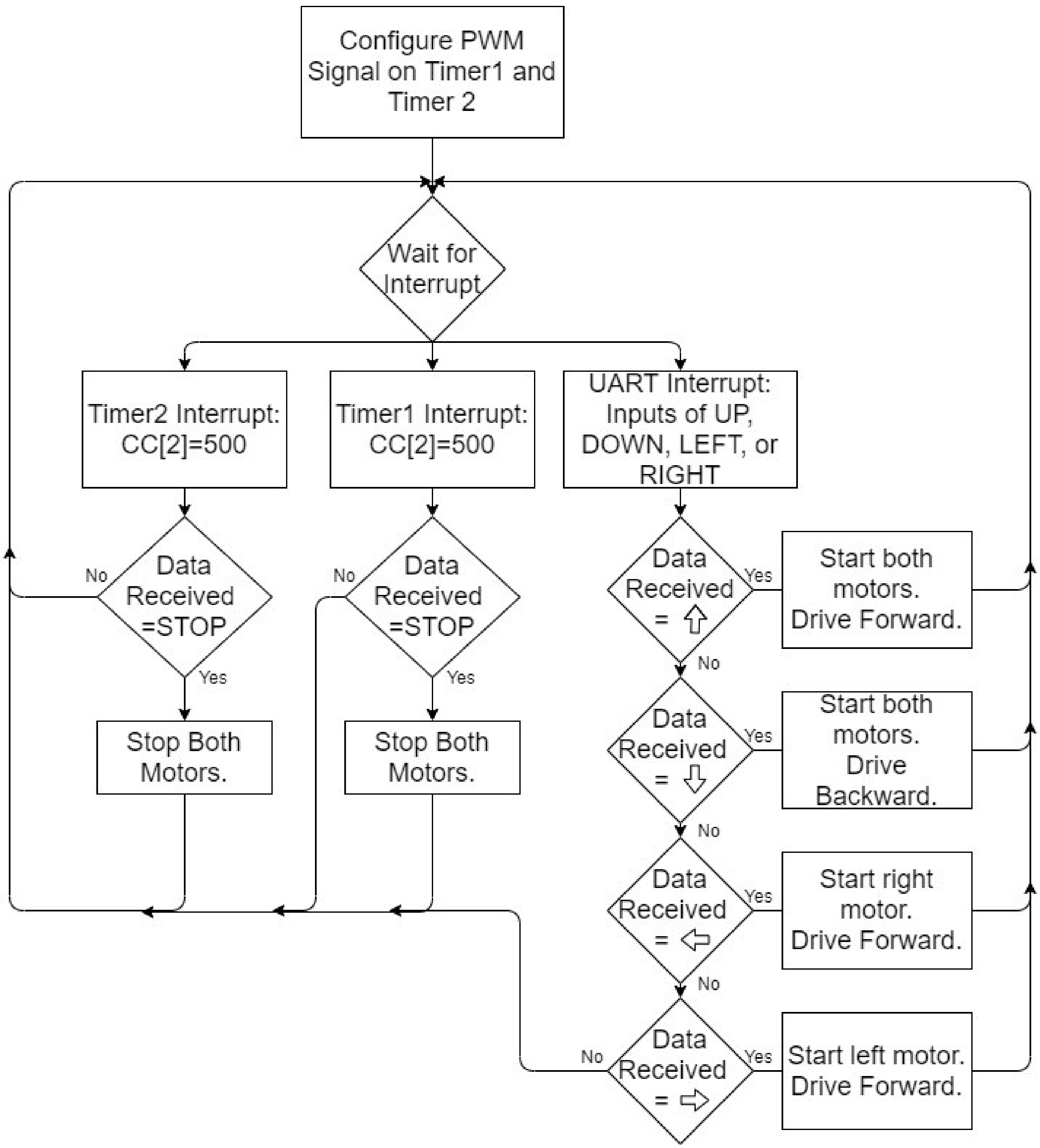

Figure 20: Microcontroller flowchart of motor operation. (To be updated for continuous input for operation of motors)

With the newer nRF52 microcontroller, an onboard PWM peripheral was ready to be used. This peripheral made it much easier to change the duty cycle. This peripheral is made to change the duty cycle without the user needing to check the current state of the signal to ensure it is in the correct state. The resulting PWM signal from this peripheral can be seen in Figure 21 and can be compared to the signal produced by the nRF51.

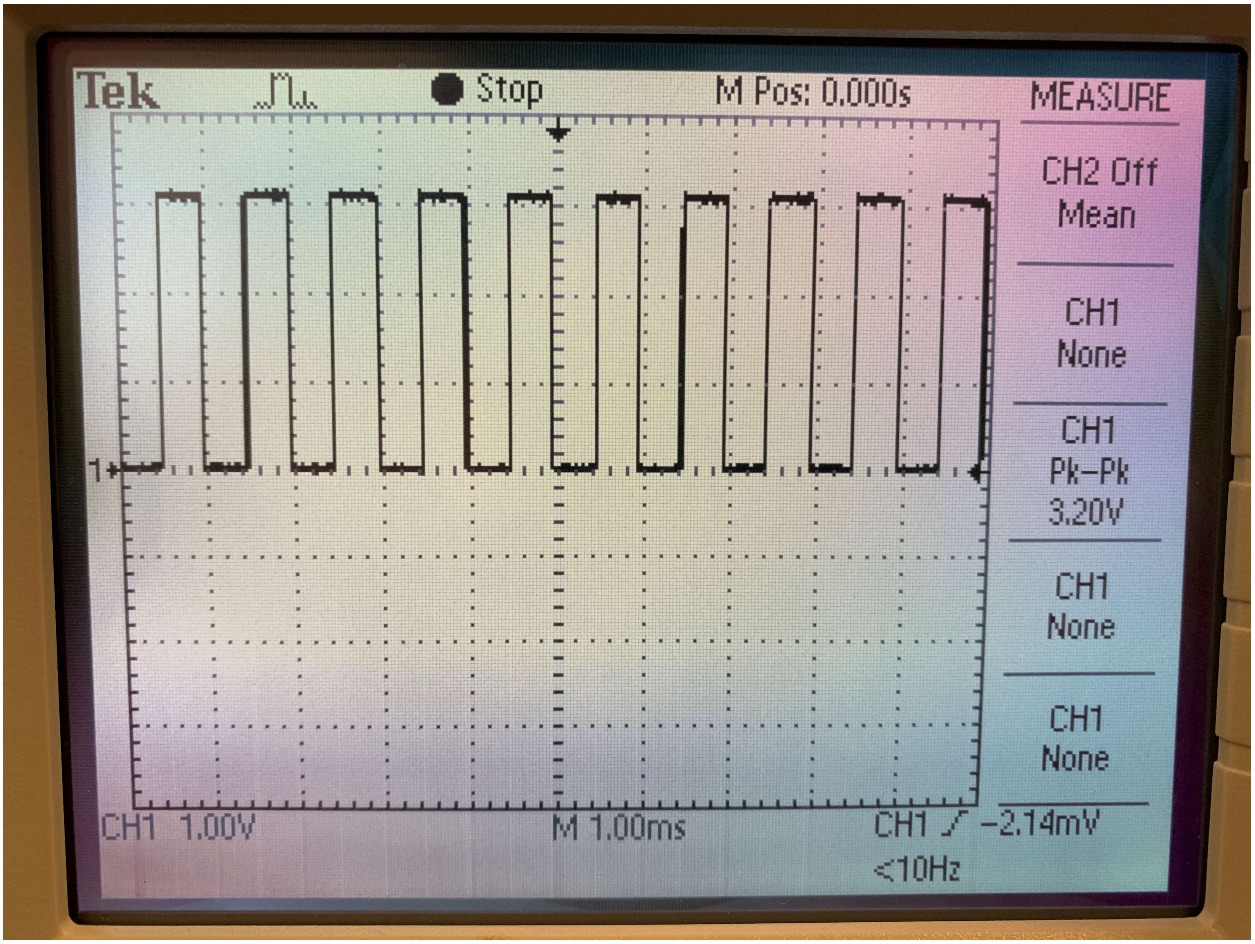

Figure 21: PWM signal generated on nRF52 microcontroller.

5.1.3 Navigation

The program was written so that the microcontroller stays in the low-power wait-for-interrupt mode until it is given an input. With the nRF51 microcontroller, the system was designed with a serial input from the universal asynchronous receiver/transmitter (UART). This input was UP, DOWN, LEFT, or RIGHT. When an interrupt occurs on the UART peripheral the signal is read, and the corresponding motors are started to drive the wheelchair in that direction. This can be seen in Figure 20.

In the updated system on the nRF52 microcontroller, the UART was removed and instead the inputs were transmitted wirelessly via the Bluetooth connection to another nRF52 microcontroller.

5.2 NORDIC S140 SOFTDEVICE

The Nordic SoftDevices are precompiled binary files which are included for ease of use in setting up the Bluetooth stack while working with the software development kit. The SoftDevice programmed to the NRF52840 was the S140. This file enables the board to act in a central or a peripheral role with up to 20 roles. A basic diagram showing how this connection is established and operates can be seen in Figure 22.

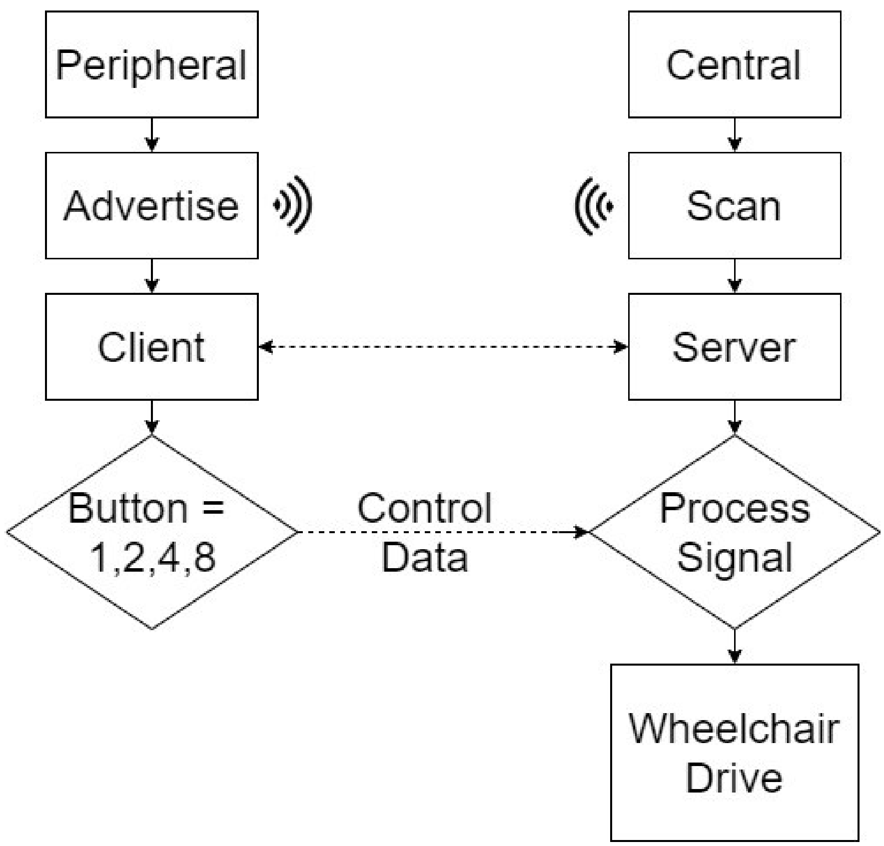

Figure 22: Bluetooth Connection Flowchart.

This leaves the flexibility to add further sensors which can control the wheelchair wirelessly. Some opportunities to implement these sensors would be proximity sensors to ensure no collisions can take place, as well as cliff sensors to scan the path for obstacles such as curbs or stairs. Further, more human sensors could be applied to the system. Specialized EMG sensors could be implemented to measure facial or other muscular movement for further control. The Bluetooth system architecture can be seen in Figure 23.

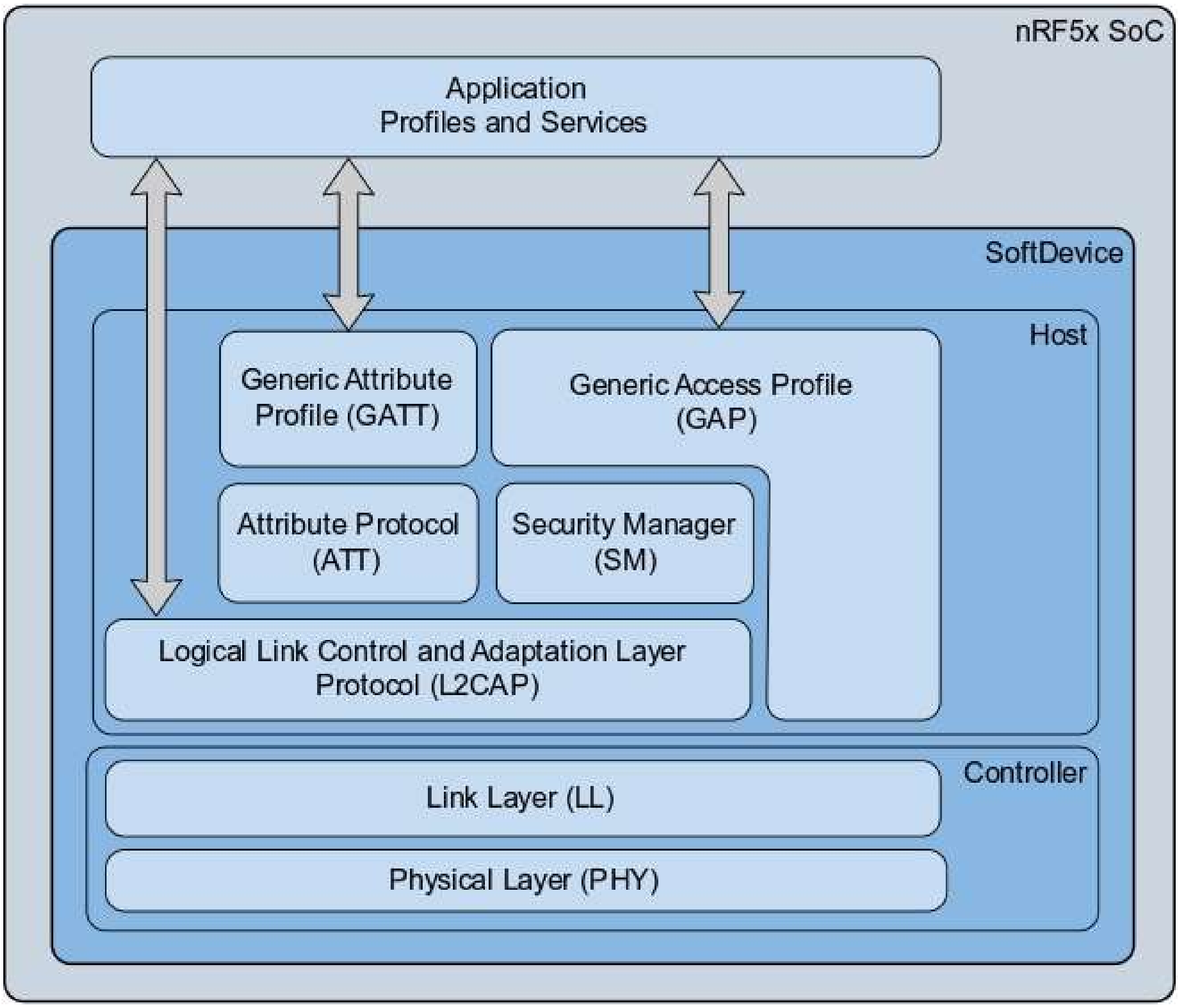

Figure 23: SoftDevice architecture. **[10]**

The Bluetooth 5.1 specification that this SoftDevice adheres to contains support for LE Data Packet Length Extension.

5.2.1 Physical Layer (PHY)

The physical layer in the Bluetooth 5.1 protocol stack enables the Bluetooth to operate in one of three ways. The first is Bluetooth LE 1M. This method of connection offers data exchange rates of 1Mbps (mega bit per second) and was introduced in Bluetooth 4.0. The second method of connection, LE 2M, offers data exchange rates of 2Mbps. Devices created before the appearance of LE 2M require that the Bluetooth specification continue to support these devices connected via the LE 1M specification. The final method of connection, 4x range, increases the device's ability to successfully transmit data over longer distances. [11]

All three of these options continue to serve specific purposes. As the amount of data being transmitted increases, the range that it can be transmitted decreases. Using the Host Controller Interface, the data to be transmitted at a given time may be changed from one method of connection to another. For example, a device may be advertising a certain signal at long range, but when a device connects with it, it may switch to a short-range, high bit rate data transfer.

The EEG headset has a data payload of 44 bytes per data packet. The headset takes and sends a reading at 250Hz. This gives a data rate of 88kbps, which is easily achievable with any of the Bluetooth connections described in this specification. The primary reason to switch to the upgraded NRF52 series chip is the LE Data Packet Length Extension. This allows the chip to send data packets up to 255 bytes in length per cycle, while the previous Bluetooth versions and thus previous chips, only supported data packets that are 20 bytes or fewer. The same goal could be accomplished with this chip but would require sending 3 packets per data set. The first packet being 20 bytes, the second packet being 20 bytes, and the final packet would contain four bytes. With the current NRF52 series chips, all 44 bytes of data can be sent in one packet.

5.2.2 Link Layer (LL)

This layer is where the type of connection is defined. How two or more devices are going to connect and/or communicate with each other will be defined here as master/slave or central/peripheral. [12]

This layer also controls the algorithm with which the devices will be changing frequencies while communicating. Because Bluetooth operates between 2.40 GHz and 2.48 GHz, there are only 40 available 2 MHz channels to be selected. In areas with high RF traffic, devices would quickly run out of empty channels to use for data transmission and would therefore be required to operate on the same channel as other unrelated devices. With these channel selection algorithms, the connected devices will have a chosen path that is determined only within that network. These devices will then change between channels at the same time as all other devices in that network.

This enables multiple devices to operate in the same area without the concern of data being lost or jumbled. [12]

The algorithm used for frequency hopping has been updated in Bluetooth 5.0 to the *channel selection algorithm #2.* The first algorithm for frequency hopping switched between one of 12 patterns. This new algorithm has switched to a pseudo random path through the optional frequencies which yields a much larger number of frequency patterns. [11]

5.2.3 Security Manager (SM)

This layer is where a link is set up between two or more devices. Here, encryption keys are created exchanged and checked between the devices to ensure that only the correct data is transferred. [12]

5.2.4 Logical Link Control Adaptation Layer (L2CAP)

The Logical Link Control Adaptation Layer is where the lower layers, physical and baseband, communicates with the upper layers, profiles, and applications. One such function performed here is packet segmentation and assembly.

5.2.5 Attribute Protocol (ATT)

The attribute protocol is where data is stored in a Bluetooth network. Pieces of information are stored as attributes in the client and server ATT protocol. The ATT consists of four fields: attribute type, attribute handle, attribute permissions, and attribute value. Attributes communicate with each other to determine which data should be transmitted/received.

5.2.6 Generic Attribute Protocol (GATT)

The generic attribute protocol is where the primary data transfer is taking place in the Bluetooth protocol. It is critical that this layer operate according to the Bluetooth Core Specification because this is what enables Bluetooth compatible devices from all around the world to interact with each other.

5.2.7 Generic Access Profile (GAP)

The generic access profile controls the connection functionality of a device. It uses the data from other layers to control how the device will function including the device discovery, connection modes, security, authentication, association models and service discovery. This is the basic information that forms a functional Bluetooth device. [13]

5.2.8 Applications and Profiles

The current state of the project has the central device set up to include a characteristic for the state of a button on the peripheral device. This characteristic uses the four least significant bits to control the operations of the wheelchair. A diagram showing the location of the bits can be seen in Figure 24.

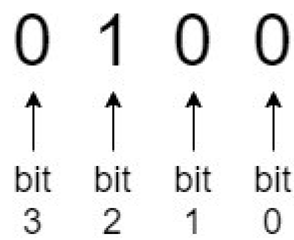

Figure 24: Four bits to control motor function in central device. In this example, bit two is on and the wheelchair would turn left.

Bit zero is the forward drive operation, bit one is the reverse operation, bit two is the left turn operation, and bit three is the right turn operation. If all the bits are set to zero, the stop operation is executed.

6 WHEELCHAIR

The wheelchair used in this project is the ActiveCare Medical, Medalist Power Wheelchair. This wheelchair was donated to the University of Southern Indiana for the previous team's project by a local wheelchair repair company, Custom Cycle and Mobility. [2]

The control system that is initially installed on the wheelchair was a multi-directional joystick which included acceleration and navigation control. A diagram of the control system can be seen in Figure 25.

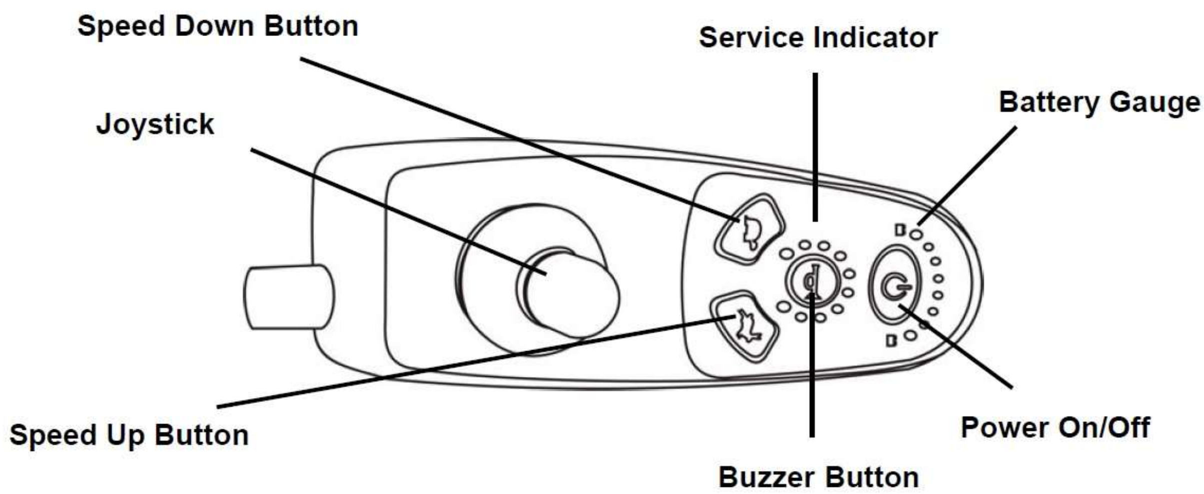

Figure 25: Wheelchair Control System. **[14]**

6.1 ELECTRONICS ORGANIZATION

Because this project began from the end point of another team's project, there was a transitioning period where the new team had to evaluate the current condition of the wheelchair. This included a complete overhaul of the device layout as well as the wiring of each device. The original design didn't use proper cable management, and it would have been difficult to reassemble the wheelchair in its condition. This was remedied with proper connectors and a more methodical organization approach. This made it simpler to continually make small changes to the system when testing new features. The before and after images can be seen in Figure 26.

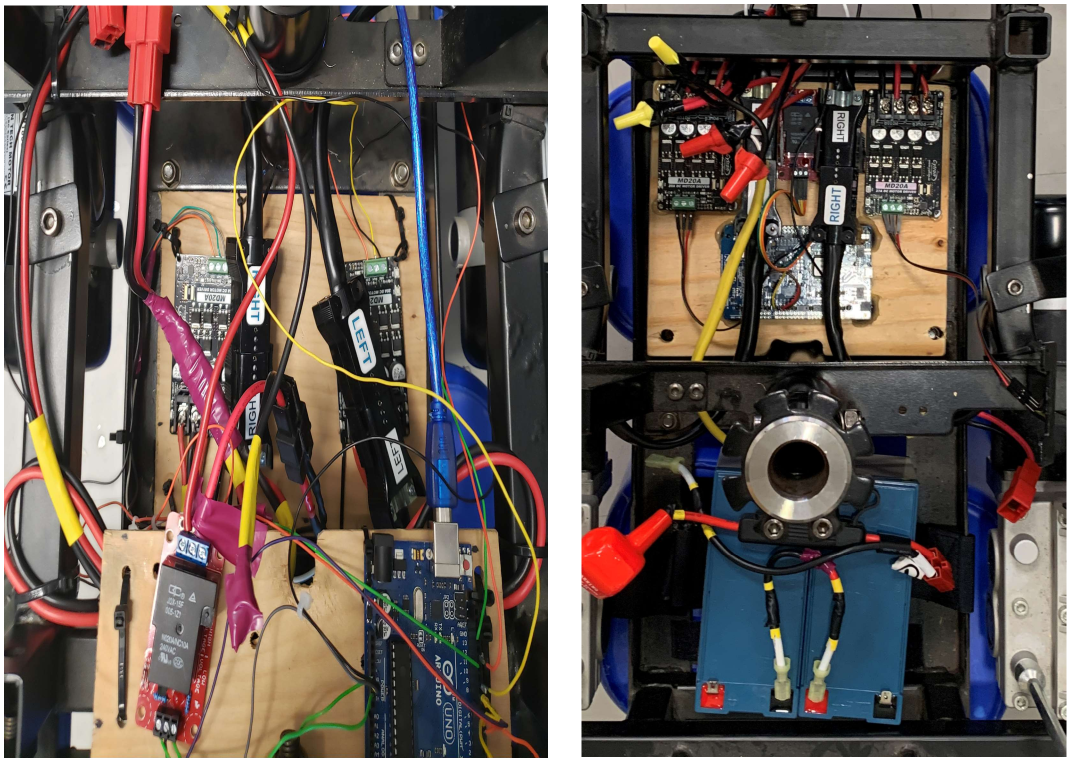

Figure 26: Before (left) and After (right) photos of the wiring of the wheelchair.

6.2 Controller

For testing purposes, the Arduino Uno used by the previous team was replaced with a Nordic NRF51 Development Board. The positioning of the new board can be seen in the after photo in Figure 26. The board was programmed using C language and a test program was written using the data received through the Universal Asynchronous Receiver Transmitter (UART) to enter conditionals that control motor movement based on data received. The code for this can be found in Appendix F.

6.3 Batteries

The wheelchair is powered by two 12 VDC x 36 Ah lead-acid batteries connected in series. These used batteries donated by CenterPoint Energy.

6.4 Motor Drivers

The motor drivers used for the wheelchair motors are the MD20A by Cytron Technologies. The location of the drivers on the wheelchair can be seen in Figure 27. These drivers are capable of supply up to 20 A each, which is more than enough for the motors on this wheelchair. The motor drivers are receiving the pulse width modulated signal from the microcontroller at approximately 3 V. This signal is then transferred to the 24 VDC from the batteries and sent to the motors.

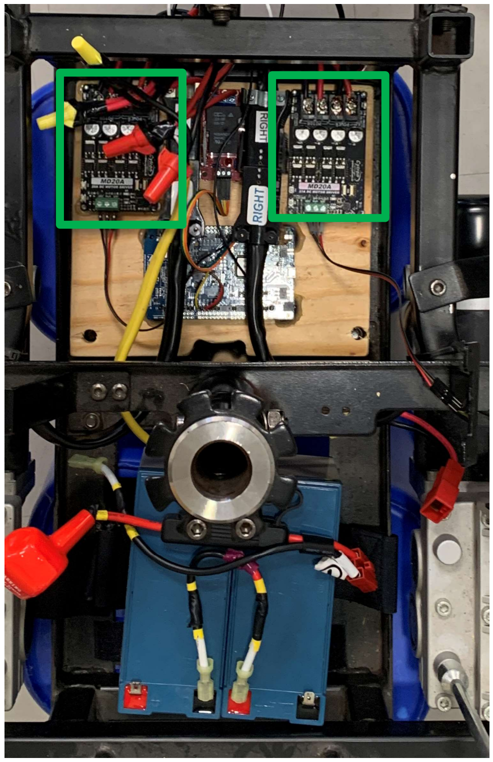

Figure 27: Motor Drivers (outlined in green)

6.5 MOTORS

The motors on the wheelchair are 24 VDC. They are rated to operate at 320 W, and a no-load speed of 4,600 RPM. The motor location on the wheelchair can been seen in Figure 28.

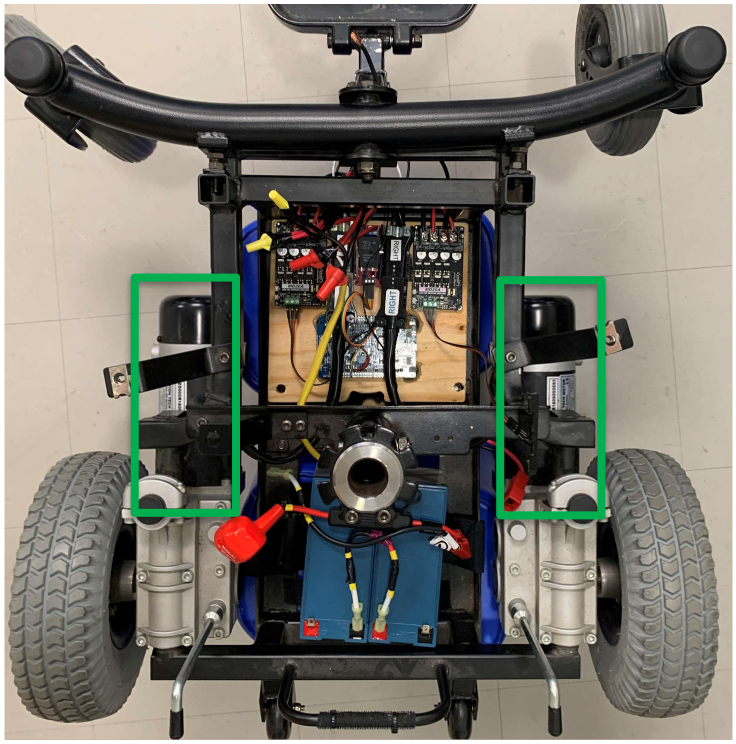

Figure 28: Wheelchair Motors (outlined in green)

7 CONSIDERATIONS

7.1 Major Considerations in Design

Some considerations are more applicable to certain projects and this wheelchair system is no exception. Because this product is a medical device, the main focus has been and always should be on helping people, which is why public health, safety, and welfare are so important.

7.1.1 Public Health, Safety, and Welfare

The product must include features to protect the user. The features may include proximity sensors to detect obstacle and stop the chair, system status indicators such as low battery warnings, and a constant input requirement where the chair will only move when the user is actively giving the signal to move.

7.1.2 Social and Cultural

This device aims to increase freedom of movement for individuals with extreme mobility limitations. There will always be some level of risk that a product malfunction, even as simple as a dead battery, could cause significant problems for these individuals who may have a difficult time getting help if they have a product malfunction. Individual users would need to weigh any potential for injury against the increased freedom of movement to determine if the product would be a good fit for them. Many people with mobility issues use different types of assistive devices such as canes, walkers, manual wheelchairs, joystick controlled electric wheelchairs, and prosthetic devices. Increasing quality of life through freedom of movement seems to be an important and well accepted goal in the medical community.

7.2 OTHER CONSIDERATIONS

7.2.1 Economic Considerations

This product will be very expensive to make. There is not likely to be a large demand meaning the production costs will be high. If the wheelchair is proven to function well it is likely that medical insurances would eventually cover at least part of the cost. While potentially expensive, the improvement in quality of life for people who need to use this chair would be an enormous benefit.

7.2.2 Environmental

The EEG controlled electric wheelchair is a very specialized product and as such will have a much smaller consumer demand. While some components, such as batteries, may need replaced the wheelchair itself should last for many years. Being able to use the product for many years while producing fewer products than most other consumer goods mean the product will have less negative impact on the environment than most consumer products.

7.2.3 Global/Political

This is a medical assistive device so it would have to get approval as a medical device. In the United States the U.S. Food and Drug Administration (FDA) regulates medical devices. Other countries are likely to have similar organizations to regulate their medical devices and the EEG controlled wheelchair may need to be submitted for approval in each country where it is to be distributed.

7.2.4 Teamwork

Multiple communication methods were explored early in the design process. The team can communicate via phone calls, text, email and zoom. A shared OneDrive folder is used to store all working documents for the projects so all team members can access the data whenever they need to. An availability calendar was created, and meetings occurred several times a week for an hour to two hours.

The tasks were divided among the team members to best fit individual strengths. Current tasks were discussed and updated as needed. Division of labor was strongly encouraged for this project, but with difficult concepts it would have been useful to have another person familiar enough with the specific concept to be able to work together to sort out issues.

When differences of opinion occur between team members several conflict resolution techniques are used. The team will discuss the pros and cons of each option. If time allows multiple options may be tested to determine which option works best. If one team member wants to add something additional to the project, they will be the one to develop the option and demonstrate the superiority to the previous design. If all members agree the design change is an improvement, the changes will be adopted. If not, the process may be repeated.

Overall, this team functioned very effectively with very few conflicts. Defining specific goals early in the process and communicating problems as they occurred helped to keep team members on track and working toward a common goal.

8 CONCLUSION AND RECOMMENDATIONS

The team was able to achieve the objective of wirelessly controlling an electric wheelchair using EEG signals. All four directional control functions (forward, backward, left, and right) are programmed onto the Bluetooth receiver. The functions turn on the appropriate motors in the correct direction to achieve the desired directional movement determined by the input received from the Bluetooth transmitter. With button presses on the Bluetooth transmitter all four directional movements were completed with the wheelchair.

A single blink was used as a stimulus event to trigger the wheelchair to move forward. MatLab was used to collect data from the headset through Simulink. It was averaged over a short time and compared to a threshold to determine if the blink stimulus occurred. A value was sent to an Arduino indicating if the threshold was met or not. A wired connection was used to connect the Arduino to the Bluetooth transmitter and the value sent to the Arduino replaced the forward button used in the earlier trials. This process allowed a blink to trigger the wheelchair to move forward with the wheelchair stopping when the averages from the EEG data normalized to below the threshold again. The complete modified system diagram can be seen in Figure 29.

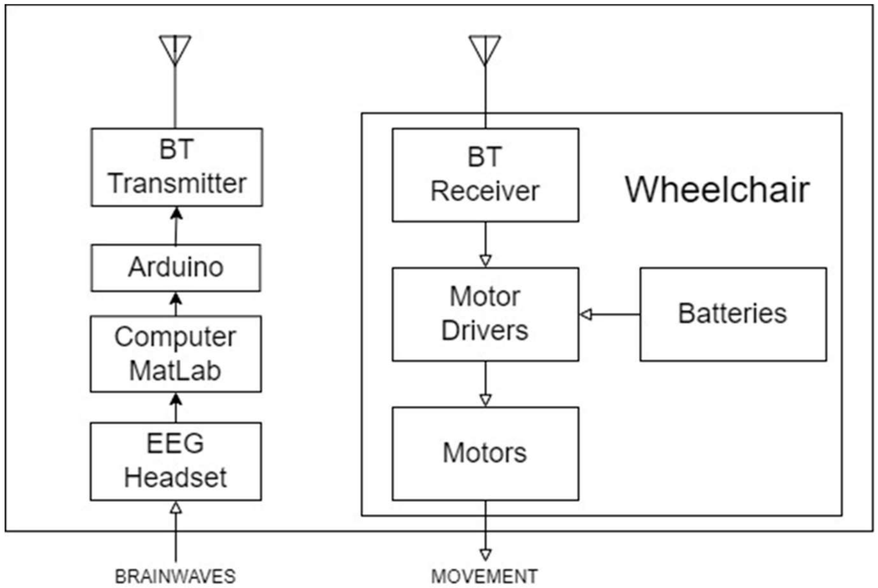

Figure 29: Final Design Block Diagram

More complete study of the data transmitted by the EEG headset will need to be completed to determine how the data is structured. Multiple tests will need to be completed to find stimuli that produce specific, recognizable, repeatable responses. Once these stimuli are found they can be used as direction indicators. Conditional statements can be used to trigger the directional control functions on the microcontroller onboard the wheelchair. Ideally, the motors should stop if a direction indicator is not present. The user should not have to send a signal to stop, the chair should only move when the user is actively sending a signal to move. Speed control is also an idea that needs further development. It would be helpful if the wheelchair would gradually increase speed when a direction indicator is received and gradually decrease speed when the direction indicator is no longer being received.

REFERENCES

[1] National Spinal Cord Injury Statistical Center, "Facts and Figures at a Glance," University of Alabama at Birmingham, Birmingham, AL, 2020.

[2] R. Blake, C. Lutz and C. (. Post, "Electroencephologram Controlled Wheel Chair," 2019.

[3] g.tec neurotechnology GmbH Austria, "User Manual for Unicorn Brain Interface Hybrid Black," g.tec neurotechnology, Austria, 2019.

[4] A. Delorme and S. Makeig, "Welcome to the EEGLAB tutorial," [Online]. Available: https://eeglab.org/tutorials/.

[5] D. Rotier, X. Zhang, Q. Guo and L. Yuan, "Research on Brain Control Technology for Wheelchair," *MATEC Web of Conferences,* vol. 232, 2018.

[6] i. u. wavelet_spaghetti, "Illumino Brainlight: Turn Your Brainwaves Into Light," 2014.

[7] instructables user 439128238, "Mindwave Wheelchair," 2020.

[8] N. Shinde and K. George, "Brain-Controlled Driving Aid for Electric Wheelchairs," *IEEE Xplore,* pp. 115-118, 2016.

[9] S. K. Swee, K. D. T. Kiang and L. Z. You, "EEG Controlled Wheelchair," *MATEC Web of Conferences,* no. 51, 2016.

[10] "Nordic Semiconductor," 06 12 2021. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_nrf52%2Fstruct%2Fnrf52.html.

[11] M. Woolley, "Bluetooth 5 / Go Faster. Go Further.," Bluetooth SIG, 2019.

[12] P. McDermott-Wells, "What is Bluetooth?," *IEEE Potentials,* pp. 33-35, 2004.

[13] "Vol. 1: Architecture, Mixing, and Conventions," in *Bluetooth Core Specification*, Bluetooth SIG, 2019.

[14] *Medalist Power Wheelchair Owner's Manual.*

[15] G. Peng, Y. Wang and R. Kasuganti, "Technological embeddedness and household computer adoption," *Information Technology & People,* vol. 24, no. 4, pp. 414-436, 2011.

[16] A. Maksud, R. I. Chowdhury, T. T. Chowdhury, S. A. Fattah, C. Shahnanaz and S. S. Chowdhury, "Low-cost EEG Based Electric Wheelchair with Advanced Control Features," in *IEE Region 10 Conference (TENCON)*, Malaysia, 2017.

[17] E. S. de Souza, E. E. Lamounier and A. A. Cardoso, "A Virtual Environment-based Training System for the Blind," *PeerJ PrePrints,* 2017.

[18] Unicorn the Brain Interface, "Unicorn Suite," Unicorn, 2020. [Online]. Available: https://www.unicorn-bi.com/unicorn-suite/. [Accessed 28 March 2021].

[19] Unicorn The Brain Interface, "Unicorn Hybrid Black for Students," Unicorn, 2020. [Online]. Available: https://www.unicorn-bi.com/unicorn-hybrid-black-for-students/. [Accessed 28 March 2021].

[20] Parallax Incorporated, "Store: 7.2V Motor, Bracket and Wheel Kit," [Online]. Available: http://www.parallax.com/Store/Robots/AllRobots/tabid/755/ProductID/587/List/0/Default.aspx?SortField=ProductName,ProductName. [Accessed 25 September 2011].

[21] Emotiv, "Headsets - EMOTIV EPOC X 14 Channel Mobile Brainwear®- Description," Emotiv, 2021. [Online]. Available: https://www.emotiv.com/product/emotiv-epoc-x-14-channel-mobile-brainwear/#tab-description. [Accessed 28 March 2021].

[22] Emotiv, "Emotivpro," Emotiv, 2021. [Online]. Available: https://www.emotiv.com/emotivpro/. [Accessed 28 March 2021].

[23] NeuroSky, "EEG Biosensors - Biosensors/EEG Headsets," NeuroSky, 2021. [Online]. Available: http://neurosky.com/biosensors/eeg-sensor/biosensors/. [Accessed 28 March 2021].

[24] NeuroSky, "EEG Biosensors - Biometric Algorithms," NeuroSky, 2021. [Online]. Available: http://neurosky.com/biosensors/eeg-sensor/algorithms/. [Accessed 28 March 2021].

APPENDIX

Appendix A: ABET Outcome 2, Design Factor Considerations

ABET Outcome 2 states "*An ability to apply engineering design to produce solutions that meet specified needs with consideration of public health safety, and welfare, as well as global, cultural, social, environmental, and economic factors.*"

ABET also requires that design projects reference appropriate professional standards, such as IEEE, ATSM, etc.

For each of the factors in Table A.1, indicate the page number(s) of your report where the item is addressed, or provide a statement regarding why the factor is not applicable for this project.

Table A.1, Design Factors Considered

Design Factor	Page number, or reason not applicable
Public health safety, and welfare	1, 30
Global	31
Cultural	30
Social	1, 30
Environmental	31
Economic	31
Professional Standards	23, 24, 25, Bluetooth Specifications

Appendix B: ABET Outcome 2, Design Factor Considerations

Bill of Materials			
Product	Cost	Qty.	Subtotal
Unicorn Hybrid Black EEG Headset*	$ 1,194.07	1	$ 1,194.07
Nordic NRF52 DK	$ 49.00	2	$ 98.00
MD20A 20Amp 6V-30V DC Motor Driver	$ 17.25	2	$ 34.50
SparkFun Beefcake Relay Control Kit (Ver. 2.0)	$ 8.95	1	$ 8.95
ActiveCare Medical: Medalist Power Wheelchair	$ 1,899.00	1	$ 1,899.00
TOTAL COST			$ 3,234.52

Appendix C: Central Device Main Code

```
/**
 * Copyright (c) 2014 - 2021, Nordic Semiconductor ASA
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice, this
 *    list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form, except as embedded into a Nordic
 *    Semiconductor ASA integrated circuit in a product or a software update for
 *    such product, must reproduce the above copyright notice, this list of
 *    conditions and the following disclaimer in the documentation and/or other
 *    materials provided with the distribution.
 *
 * 3. Neither the name of Nordic Semiconductor ASA nor the names of its
 *    contributors may be used to endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * 4. This software, with or without modification, must only be used with a
 *    Nordic Semiconductor ASA integrated circuit.
 *
 * 5. Any software provided in binary form under this license must not be reverse
 *    engineered, decompiled, modified and/or disassembled.
 *
 * THIS SOFTWARE IS PROVIDED BY NORDIC SEMICONDUCTOR ASA "AS IS" AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL NORDIC SEMICONDUCTOR ASA OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */
/**
 * @brief BLE LED Button Service central and client application main file.
 *
 * This file contains the source code for a sample client application using the LED Button service.
 */

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "nrf.h" //pwm
#include "nrf_gpio.h"
#include "nrf_sdh.h"
```

```
#include "nrf_sdh_ble.h"
#include "nrf_sdh_soc.h"
#include "nrf_pwr_mgmt.h"
#include "app_timer.h"
#include "boards.h"
#include "bsp.h"
#include "bsp_btn_ble.h"
#include "ble.h"
#include "ble_hci.h"
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "ble_db_discovery.h"
#include "ble_lbs_c.h"
#include "nrf_ble_gatt.h"
#include "nrf_ble_scan.h"

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"

#include <stdbool.h>
//#include <stdint.h>

#include "app_error.h"
//#include "bsp.h"
#include "nrf_delay.h"
#include "app_pwm.h"


#define CENTRAL_SCANNING_LED          BSP_BOARD_LED_0                /**< Scanning LED will be on when the
device is scanning. */
#define CENTRAL_CONNECTED_LED         BSP_BOARD_LED_1                /**< Connected LED will be on when the
device is connected. */
#define LEDBUTTON_LED                 BSP_BOARD_LED_2                /**< LED to indicate a change of state of the
the Button characteristic on the peer. */

#define SCAN_INTERVAL                 0x00A0                         /**< Determines scan interval in units of 0.625
millisecond. */
#define SCAN_WINDOW                   0x0050                         /**< Determines scan window in units of 0.625
millisecond. */
#define SCAN_DURATION                 0x0000                         /**< Timout when scanning. 0x0000 disables timeout.
*/

#define MIN_CONNECTION_INTERVAL       MSEC_TO_UNITS(7.5, UNIT_1_25_MS)    /**< Determines minimum
connection interval in milliseconds. */
#define MAX_CONNECTION_INTERVAL       MSEC_TO_UNITS(30, UNIT_1_25_MS)     /**< Determines maximum
connection interval in milliseconds. */
#define SLAVE_LATENCY                 0                              /**< Determines slave latency in terms of connection
events. */
#define SUPERVISION_TIMEOUT           MSEC_TO_UNITS(4000, UNIT_10_MS)     /**< Determines supervision time-
out in units of 10 milliseconds. */
```

```
#define LEDBUTTON_BUTTON_PIN           BSP_BUTTON_0                   /**< Button that will write to the LED
characteristic of the peer */
#define BUTTON_DETECTION_DELAY         APP_TIMER_TICKS(50)              /**< Delay from a GPIOTE event until a
button is reported as pushed (in number of timer ticks). */

#define APP_BLE_CONN_CFG_TAG           1                             /**< A tag identifying the SoftDevice BLE
configuration. */
#define APP_BLE_OBSERVER_PRIO          3                             /**< Application's BLE observer priority. You
shouldn't need to modify this value. */


APP_PWM_INSTANCE(PWM1,1);                // Create the instance "PWM1" using TIMER1.

static volatile bool ready_flag;           // A flag indicating PWM status.

void pwm_ready_callback(uint32_t pwm_id)    // PWM callback function
{
   ready_flag = true;
}

NRF_BLE_SCAN_DEF(m_scan);                                /**< Scanning module instance. */
BLE_LBS_C_DEF(m_ble_lbs_c);                              /**< Main structure used by the LBS client module. */
NRF_BLE_GATT_DEF(m_gatt);                                /**< GATT module instance. */
BLE_DB_DISCOVERY_DEF(m_db_disc);                         /**< DB discovery module instance. */
NRF_BLE_GQ_DEF(m_ble_gatt_queue,                         /**< BLE GATT Queue instance. */
            NRF_SDH_BLE_CENTRAL_LINK_COUNT,
            NRF_BLE_GQ_QUEUE_SIZE);

static char const m_target_periph_name[] = "Nordic_Blinky";     /**< Name of the device we try to connect to. This
name is searched in the scan report data*/

uint8_t data = 0;


/**@brief Function to handle asserts in the SoftDevice.
 *
 * @details This function will be called in case of an assert in the SoftDevice.
 *
 * @warning This handler is an example only and does not fit a final product. You need to analyze
 *          how your product is supposed to react in case of Assert.
 * @warning On assert from the SoftDevice, the system can only recover on reset.
 *
 * @param[in] line_num     Line number of the failing ASSERT call.
 * @param[in] p_file_name  File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
   app_error_handler(0xDEADBEEF, line_num, p_file_name);
}


/**@brief Function for handling the LED Button Service client errors.
 *
```

```
 * @param[in]   nrf_error   Error code containing information about what went wrong.
 */
static void lbs_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for the LEDs initialization.
 *
 * @details Initializes all LEDs used by the application.
 */
static void leds_init(void)
{
    bsp_board_init(BSP_INIT_LEDS);
}


/**@brief Function to start scanning.
 */
static void scan_start(void)
{
    ret_code_t err_code;

    err_code = nrf_ble_scan_start(&m_scan);
    APP_ERROR_CHECK(err_code);

    bsp_board_led_off(CENTRAL_CONNECTED_LED);
    bsp_board_led_on(CENTRAL_SCANNING_LED);
}


/**@brief Handles events coming from the LED Button central module.
 */
static void lbs_c_evt_handler(ble_lbs_c_t * p_lbs_c, ble_lbs_c_evt_t * p_lbs_c_evt)
{
    switch (p_lbs_c_evt->evt_type)
    {
        case BLE_LBS_C_EVT_DISCOVERY_COMPLETE:
        {
            ret_code_t err_code;

            err_code = ble_lbs_c_handles_assign(&m_ble_lbs_c,
                                                p_lbs_c_evt->conn_handle,
                                                &p_lbs_c_evt->params.peer_db);
            NRF_LOG_INFO("LED Button service discovered on conn_handle 0x%x.", p_lbs_c_evt->conn_handle);

            err_code = app_button_enable();
            APP_ERROR_CHECK(err_code);

            // LED Button service discovered. Enable notification of Button.
            err_code = ble_lbs_c_button_notif_enable(p_lbs_c);
            APP_ERROR_CHECK(err_code);
```

```
} break; // BLE_LBS_C_EVT_DISCOVERY_COMPLETE

case BLE_LBS_C_EVT_BUTTON_NOTIFICATION:
{
  //NRF_LOG_INFO("Button state changed on peer to 0x%x.", p_lbs_c_evt->params.button.button_state);


  //button state to control your GPIO output
  //gpio write

   //nrf_gpio_pin_write(25,0);
   //nrf_gpio_pin_write(24,0);

  if (p_lbs_c_evt->params.button.button_state==0x1)
  {
    //bsp_board_led_on(LEDBUTTON_LED);
    //NRF_LOG_INFO("Andrew says 0x1");
    //data = 0x1;
    NRF_LOG_INFO("Drive Forward");
    //motordriver pin dir
    nrf_gpio_pin_write(25,0);
    nrf_gpio_pin_write(24,0);
    app_pwm_channel_duty_set(&PWM1, 0, 40);
    app_pwm_channel_duty_set(&PWM1, 1, 40);
    //nrf_gpio_pin_write(25,0);
    //nrf_gpio_pin_write(23,1);

  }
  else if (p_lbs_c_evt->params.button.button_state==0x2)
  {
    //bsp_board_led_on(LEDBUTTON_LED);
    //NRF_LOG_INFO("Andrew says 0x2");
    //data = 0x2;
    NRF_LOG_INFO("Drive Backward");
    //motordriver pin dir
    nrf_gpio_pin_write(25,1);
    nrf_gpio_pin_write(24,1);
    app_pwm_channel_duty_set(&PWM1, 0, 40);
    app_pwm_channel_duty_set(&PWM1, 1, 40);
    //nrf_gpio_pin_write(25,0);
    //nrf_gpio_pin_write(24,0);
  }
  else if (p_lbs_c_evt->params.button.button_state==0x4)
  {
    //bsp_board_led_on(LEDBUTTON_LED);
    //NRF_LOG_INFO("Andrew says 0x4");
    //data = 0x4;
    NRF_LOG_INFO("Turn Left");
    //motordriver pin dir
    nrf_gpio_pin_write(25,0);
    nrf_gpio_pin_write(24,0);
    app_pwm_channel_duty_set(&PWM1, 0, 5);
    app_pwm_channel_duty_set(&PWM1, 1, 60);
```

```
        }
        else if (p_lbs_c_evt->params.button.button_state==0x8)
        {
          //bsp_board_led_on(LEDBUTTON_LED);
          //NRF_LOG_INFO("Andrew says 0x8");
          //data = 0x8;
          NRF_LOG_INFO("Turn Right");
          //motordriver pin dir
          nrf_gpio_pin_write(25,0);
          nrf_gpio_pin_write(24,0);
          app_pwm_channel_duty_set(&PWM1, 0, 60);
          app_pwm_channel_duty_set(&PWM1, 1, 5);

        }
        else if (p_lbs_c_evt->params.button.button_state==0x0)
        {
          //bsp_board_led_off(LEDBUTTON_LED);
          //NRF_LOG_INFO("Andrew says off");
          //data = 0x0;
          NRF_LOG_INFO("All Motors Stop");
          //motordriver pin dir
          nrf_gpio_pin_write(25,0);
          nrf_gpio_pin_write(24,0);
          app_pwm_channel_duty_set(&PWM1, 0, 0);
          app_pwm_channel_duty_set(&PWM1, 1, 0);

        }
      //if (data==0x0)
      //{
      //NRF_LOG_INFO("All Motors Stop");
      //}
      //if (data==0x1)
      //{
      //NRF_LOG_INFO("Drive Forward");
      //}
      //if (data==0x2)
      //{
      //NRF_LOG_INFO("Drive Backward");
      //}
      //if (data==0x4)
      //{
      //NRF_LOG_INFO("Turn Left");
      //}
      //if (data==0x8)
      //{
      //NRF_LOG_INFO("Turn Right");
      //}
      } break; // BLE_LBS_C_EVT_BUTTON_NOTIFICATION

      default:
        // No implementation needed.
        break;
```

```
    }
}


/**@brief Function for handling BLE events.
 *
 * @param[in]   p_ble_evt   Bluetooth stack event.
 * @param[in]   p_context   Unused.
 */
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code;

    // For readability.
    ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;

    switch (p_ble_evt->header.evt_id)
    {
        // Upon connection, check which peripheral has connected (HR or RSC), initiate DB
        // discovery, update LEDs status and resume scanning if necessary. */
        case BLE_GAP_EVT_CONNECTED:
        {
            NRF_LOG_INFO("Connected.");
            err_code = ble_lbs_c_handles_assign(&m_ble_lbs_c, p_gap_evt->conn_handle, NULL);
            APP_ERROR_CHECK(err_code);

            err_code = ble_db_discovery_start(&m_db_disc, p_gap_evt->conn_handle);
            APP_ERROR_CHECK(err_code);

            // Update LEDs status, and check if we should be looking for more
            // peripherals to connect to.
            bsp_board_led_on(CENTRAL_CONNECTED_LED);
            bsp_board_led_off(CENTRAL_SCANNING_LED);
        } break;

        // Upon disconnection, reset the connection handle of the peer which disconnected, update
        // the LEDs status and start scanning again.
        case BLE_GAP_EVT_DISCONNECTED:
        {
            app_pwm_channel_duty_set(&PWM1, 0, 0);
            app_pwm_channel_duty_set(&PWM1, 1, 0);
            NRF_LOG_INFO("Disconnected.");
            scan_start();
        } break;

        case BLE_GAP_EVT_TIMEOUT:
        {
            // We have not specified a timeout for scanning, so only connection attemps can timeout.
            if (p_gap_evt->params.timeout.src == BLE_GAP_TIMEOUT_SRC_CONN)
            {
                NRF_LOG_DEBUG("Connection request timed out.");
            }
        } break;
```

```
        case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
        {
            // Accept parameters requested by peer.
            err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle,
                                   &p_gap_evt->params.conn_param_update_request.conn_params);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
        {
            NRF_LOG_DEBUG("PHY update request.");
            ble_gap_phys_t const phys =
            {
                .rx_phys = BLE_GAP_PHY_AUTO,
                .tx_phys = BLE_GAP_PHY_AUTO,
            };
            err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GATTC_EVT_TIMEOUT:
        {
            // Disconnect on GATT Client timeout event.
            NRF_LOG_DEBUG("GATT Client Timeout.");
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                                   BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GATTS_EVT_TIMEOUT:
        {
            // Disconnect on GATT Server timeout event.
            NRF_LOG_DEBUG("GATT Server Timeout.");
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                                   BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
        } break;

        default:
            // No implementation needed.
            break;
    }
}


/**@brief LED Button client initialization.
 */
static void lbs_c_init(void)
{
    ret_code_t       err_code;
    ble_lbs_c_init_t lbs_c_init_obj;
```

```
    lbs_c_init_obj.evt_handler   = lbs_c_evt_handler;
    lbs_c_init_obj.p_gatt_queue  = &m_ble_gatt_queue;
    lbs_c_init_obj.error_handler = lbs_error_handler;

    err_code = ble_lbs_c_init(&m_ble_lbs_c, &lbs_c_init_obj);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupts.
 */
static void ble_stack_init(void)
{
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}


/**@brief Function for handling events from the button handler module.
 *
 * @param[in] pin_no        The pin that the event applies to.
 * @param[in] button_action The button action (press/release).
 */
static void button_event_handler(uint8_t pin_no, uint8_t button_action)
{
    ret_code_t err_code;

    switch (pin_no)
    {
        case LEDBUTTON_BUTTON_PIN:
            err_code = ble_lbs_led_status_send(&m_ble_lbs_c, button_action);
            if (err_code != NRF_SUCCESS &&
                err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
                err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
```

```
        }
        if (err_code == NRF_SUCCESS)
        {
            NRF_LOG_INFO("LBS write LED state %d", button_action);
        }
        break;

    default:
        APP_ERROR_HANDLER(pin_no);
        break;
    }
}


/**@brief Function for handling Scaning events.
 *
 * @param[in]   p_scan_evt   Scanning event.
 */
static void scan_evt_handler(scan_evt_t const * p_scan_evt)
{
    ret_code_t err_code;

    switch(p_scan_evt->scan_evt_id)
    {
        case NRF_BLE_SCAN_EVT_CONNECTING_ERROR:
            err_code = p_scan_evt->params.connecting_err.err_code;
            APP_ERROR_CHECK(err_code);
            break;
        default:
          break;
    }
}



/**@brief Function for initializing the button handler module.
 */
static void buttons_init(void)
{
    ret_code_t err_code;

    //The array must be static because a pointer to it will be saved in the button handler module.
    static app_button_cfg_t buttons[] =
    {
        {LEDBUTTON_BUTTON_PIN, false, BUTTON_PULL, button_event_handler}
    };

    err_code = app_button_init(buttons, ARRAY_SIZE(buttons),
                               BUTTON_DETECTION_DELAY);
    APP_ERROR_CHECK(err_code);
}
```

```
/**@brief Function for handling database discovery events.
 *
 * @details This function is callback function to handle events from the database discovery module.
 *          Depending on the UUIDs that are discovered, this function should forward the events
 *          to their respective services.
 *
 * @param[in] p_event  Pointer to the database discovery event.
 */
static void db_disc_handler(ble_db_discovery_evt_t * p_evt)
{
    ble_lbs_on_db_disc_evt(&m_ble_lbs_c, p_evt);
}


/**@brief Database discovery initialization.
 */
static void db_discovery_init(void)
{
    ble_db_discovery_init_t db_init;

    memset(&db_init, 0, sizeof(db_init));

    db_init.evt_handler  = db_disc_handler;
    db_init.p_gatt_queue = &m_ble_gatt_queue;

    ret_code_t err_code = ble_db_discovery_init(&db_init);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the log.
 */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}


/**@brief Function for initializing the timer.
 */
static void timer_init(void)
{
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the Power manager. */
static void power_management_init(void)
{
```

```
   ret_code_t err_code;
   err_code = nrf_pwr_mgmt_init();
   APP_ERROR_CHECK(err_code);
}


static void scan_init(void)
{
   ret_code_t          err_code;
   nrf_ble_scan_init_t init_scan;

   memset(&init_scan, 0, sizeof(init_scan));

   init_scan.connect_if_match = true;
   init_scan.conn_cfg_tag     = APP_BLE_CONN_CFG_TAG;

   err_code = nrf_ble_scan_init(&m_scan, &init_scan, scan_evt_handler);
   APP_ERROR_CHECK(err_code);

   // Setting filters for scanning.
   err_code = nrf_ble_scan_filters_enable(&m_scan, NRF_BLE_SCAN_NAME_FILTER, false);
   APP_ERROR_CHECK(err_code);

   err_code = nrf_ble_scan_filter_set(&m_scan, SCAN_NAME_FILTER, m_target_periph_name);
   APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the GATT module.
 */
static void gatt_init(void)
{
   ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
   APP_ERROR_CHECK(err_code);
}

/**@brief Function for handling the idle state (main loop).
 *
 * @details Handle any pending log operation(s), then sleep until the next event occurs.
 */
static void idle_state_handle(void)
{
   NRF_LOG_FLUSH();
   nrf_pwr_mgmt_run();
}

static void gpio_init()
{
   nrf_gpio_cfg(25,1,1,0,0,0);
   nrf_gpio_cfg(24,1,1,0,0,0);
   nrf_gpio_cfg(21,1,1,0,0,0);
   //nrf_gpio_pin_write(25,1);
   nrf_gpio_pin_write(21,0);
```

```
}

static void pwm_init()
{
   ret_code_t err_code;

   /* 2-channel PWM, 200Hz, output on DK LED pins. */
   app_pwm_config_t pwm1_cfg = APP_PWM_DEFAULT_CONFIG_2CH(800L, 26, 27);

   /* Switch the polarity of the second channel. */
   pwm1_cfg.pin_polarity[0] = APP_PWM_POLARITY_ACTIVE_HIGH;
   pwm1_cfg.pin_polarity[1] = APP_PWM_POLARITY_ACTIVE_HIGH;

   /* Initialize and enable PWM. */
   err_code = app_pwm_init(&PWM1,&pwm1_cfg,pwm_ready_callback);
   APP_ERROR_CHECK(err_code);
   app_pwm_enable(&PWM1);
}

int main(void)
{
   // Initialize.
   log_init();
   gpio_init();
   timer_init();
   leds_init();
   buttons_init();
   power_management_init();
   ble_stack_init();
   scan_init();
   gatt_init();
   db_discovery_init();
   lbs_c_init();
   pwm_init();

   // Start execution.
   NRF_LOG_INFO("Blinky CENTRAL example started.");
   scan_start();

   // Turn on the LED to signal scanning.
   bsp_board_led_on(CENTRAL_SCANNING_LED);

   // Enter main loop.
   for (;;)
   {
      idle_state_handle();
      //while (app_pwm_channel_duty_set(&PWM1, 0, 50) == NRF_ERROR_BUSY);
     // while (app_pwm_channel_duty_set(&PWM1, 1, 10) == NRF_ERROR_BUSY);

   }
}
```

Appendix D: Peripheral Device Main Code

```
/**
 * Copyright (c) 2015 - 2021, Nordic Semiconductor ASA
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice, this
 *    list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form, except as embedded into a Nordic
 *    Semiconductor ASA integrated circuit in a product or a software update for
 *    such product, must reproduce the above copyright notice, this list of
 *    conditions and the following disclaimer in the documentation and/or other
 *    materials provided with the distribution.
 *
 * 3. Neither the name of Nordic Semiconductor ASA nor the names of its
 *    contributors may be used to endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * 4. This software, with or without modification, must only be used with a
 *    Nordic Semiconductor ASA integrated circuit.
 *
 * 5. Any software provided in binary form under this license must not be reverse
 *    engineered, decompiled, modified and/or disassembled.
 *
 * THIS SOFTWARE IS PROVIDED BY NORDIC SEMICONDUCTOR ASA "AS IS" AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL NORDIC SEMICONDUCTOR ASA OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */
/**
 * @brief Blinky Sample Application main file.
 *
 * This file contains the source code for a sample server application using the LED Button service.
 */

#include <stdint.h>
#include <string.h>
#include "nordic_common.h"
#include "nrf.h"
#include "app_error.h"
#include "ble.h"
```

```
#include "ble_err.h"
#include "ble_hci.h"
#include "ble_srv_common.h"
#include "ble_advdata.h"
#include "ble_conn_params.h"
#include "nrf_sdh.h"
#include "nrf_sdh_ble.h"
#include "boards.h"
#include "app_timer.h"
#include "app_button.h"
#include "ble_lbs.h"
#include "nrf_ble_gatt.h"
#include "nrf_ble_qwr.h"
#include "nrf_pwr_mgmt.h"

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"


#define ADVERTISING_LED             BSP_BOARD_LED_0                     /**< Is on when device is advertising. */
#define CONNECTED_LED               BSP_BOARD_LED_1                     /**< Is on when device has connected. */
#define LEDBUTTON_LED               BSP_BOARD_LED_2                     /**< LED to be toggled with the help of the
LED Button Service. */
#define LEDBUTTON_BUTTON            BSP_BUTTON_0                        /**< Button that will trigger the notification
event with the LED Button Service */

#define DEVICE_NAME                 "Nordic_Blinky"                     /**< Name of device. Will be included in the
advertising data. */

#define APP_BLE_OBSERVER_PRIO       3                                   /**< Application's BLE observer priority. You
shouldn't need to modify this value. */
#define APP_BLE_CONN_CFG_TAG        1                                   /**< A tag identifying the SoftDevice BLE
configuration. */

#define APP_ADV_INTERVAL            64                                  /**< The advertising interval (in units of 0.625 ms;
this value corresponds to 40 ms). */
#define APP_ADV_DURATION            BLE_GAP_ADV_TIMEOUT_GENERAL_UNLIMITED   /**< The advertising
time-out (in units of seconds). When set to 0, we will never time out. */


#define MIN_CONN_INTERVAL           MSEC_TO_UNITS(100, UNIT_1_25_MS)    /**< Minimum acceptable
connection interval (0.5 seconds). */
#define MAX_CONN_INTERVAL           MSEC_TO_UNITS(200, UNIT_1_25_MS)    /**< Maximum acceptable
connection interval (1 second). */
#define SLAVE_LATENCY               0                                   /**< Slave latency. */
#define CONN_SUP_TIMEOUT            MSEC_TO_UNITS(4000, UNIT_10_MS)     /**< Connection supervisory
time-out (4 seconds). */

#define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER_TICKS(20000)          /**< Time from initiating
event (connect or start of notification) to first time sd_ble_gap_conn_param_update is called (15 seconds). */
#define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(5000)           /**< Time between each call
to sd_ble_gap_conn_param_update after the first call (5 seconds). */
```

```
#define MAX_CONN_PARAMS_UPDATE_COUNT    3                               /**< Number of attempts before giving
up the connection parameter negotiation. */

#define BUTTON_DETECTION_DELAY         APP_TIMER_TICKS(50)                  /**< Delay from a GPIOTE event until
a button is reported as pushed (in number of timer ticks). */

#define DEAD_BEEF                   0xDEADBEEF                        /**< Value used as error code on stack dump, can be
used to identify stack location on stack unwind. */


BLE_LBS_DEF(m_lbs);                                                 /**< LED Button Service instance. */
NRF_BLE_GATT_DEF(m_gatt);                                             /**< GATT module instance. */
NRF_BLE_QWR_DEF(m_qwr);                                              /**< Context for the Queued Write module.*/
uint8_t data=0;
static uint16_t m_conn_handle = BLE_CONN_HANDLE_INVALID;                    /**< Handle of the current
connection. */

static uint8_t m_adv_handle = BLE_GAP_ADV_SET_HANDLE_NOT_SET;                /**< Advertising handle used to
identify an advertising set. */
static uint8_t m_enc_advdata[BLE_GAP_ADV_SET_DATA_SIZE_MAX];                 /**< Buffer for storing an
encoded advertising set. */
static uint8_t m_enc_scan_response_data[BLE_GAP_ADV_SET_DATA_SIZE_MAX];        /**< Buffer for storing an
encoded scan data. */

/**@brief Struct that contains pointers to the encoded advertising data. */
static ble_gap_adv_data_t m_adv_data =
{
    .adv_data =
    {
        .p_data = m_enc_advdata,
        .len    = BLE_GAP_ADV_SET_DATA_SIZE_MAX
    },
    .scan_rsp_data =
    {
        .p_data = m_enc_scan_response_data,
        .len    = BLE_GAP_ADV_SET_DATA_SIZE_MAX

    }
};

/**@brief Function for assert macro callback.
 *
 * @details This function will be called in case of an assert in the SoftDevice.
 *
 * @warning This handler is an example only and does not fit a final product. You need to analyze
 *          how your product is supposed to react in case of Assert.
 * @warning On assert from the SoftDevice, the system can only recover on reset.
 *
 * @param[in] line_num    Line number of the failing ASSERT call.
 * @param[in] p_file_name File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
```

```
    app_error_handler(DEAD_BEEF, line_num, p_file_name);
}


/**@brief Function for the LEDs initialization.
 *
 * @details Initializes all LEDs used by the application.
 */
static void leds_init(void)
{
    bsp_board_init(BSP_INIT_LEDS);
}


/**@brief Function for the Timer initialization.
 *
 * @details Initializes the timer module.
 */
static void timers_init(void)
{
    // Initialize timer module, making it use the scheduler
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for the GAP initialization.
 *
 * @details This function sets up all the necessary GAP (Generic Access Profile) parameters of the
 *          device including the device name, appearance, and the preferred connection parameters.
 */
static void gap_params_init(void)
{
    ret_code_t              err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                          (const uint8_t *)DEVICE_NAME,
                                          strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);

    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
```

```
}


/**@brief Function for initializing the GATT module.
 */
static void gatt_init(void)
{
 ble_gatts_char_handles_t gatts_char_handle = {0,
                                    BLE_GATT_HANDLE_INVALID,
                                    BLE_GATT_HANDLE_INVALID,
                                    BLE_GATT_HANDLE_INVALID};

   ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
   //err_code = characteristic_add(BLE_GATT_HANDLE_INVALID, 0, &gatts_char_handle);
   APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the Advertising functionality.
 *
 * @details Encodes the required advertising data and passes it to the stack.
 *          Also builds a structure to be passed to the stack when starting advertising.
 */
static void advertising_init(void)
{
   ret_code_t    err_code;
   ble_advdata_t advdata;
   ble_advdata_t srdata;

   ble_uuid_t adv_uuids[] = {{LBS_UUID_SERVICE, m_lbs.uuid_type}};

   // Build and set advertising data.
   memset(&advdata, 0, sizeof(advdata));

   advdata.name_type          = BLE_ADVDATA_FULL_NAME;
   advdata.include_appearance = true;
   advdata.flags              = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;


   memset(&srdata, 0, sizeof(srdata));
   srdata.uuids_complete.uuid_cnt = sizeof(adv_uuids) / sizeof(adv_uuids[0]);
   srdata.uuids_complete.p_uuids  = adv_uuids;

   err_code = ble_advdata_encode(&advdata, m_adv_data.adv_data.p_data, &m_adv_data.adv_data.len);
   APP_ERROR_CHECK(err_code);

   err_code = ble_advdata_encode(&srdata, m_adv_data.scan_rsp_data.p_data,
&m_adv_data.scan_rsp_data.len);
   APP_ERROR_CHECK(err_code);

   ble_gap_adv_params_t adv_params;

   // Set advertising parameters.
```

```
    memset(&adv_params, 0, sizeof(adv_params));

    adv_params.primary_phy     = BLE_GAP_PHY_1MBPS;
    adv_params.duration        = APP_ADV_DURATION;
    adv_params.properties.type = BLE_GAP_ADV_TYPE_CONNECTABLE_SCANNABLE_UNDIRECTED;
    adv_params.p_peer_addr     = NULL;
    adv_params.filter_policy   = BLE_GAP_ADV_FP_ANY;
    adv_params.interval        = APP_ADV_INTERVAL;

    err_code = sd_ble_gap_adv_set_configure(&m_adv_handle, &m_adv_data, &adv_params);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling Queued Write Module errors.
 *
 * @details A pointer to this function will be passed to each service which may need to inform the
 *          application about an error.
 *
 * @param[in]   nrf_error   Error code containing information about what went wrong.
 */
static void nrf_qwr_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for handling write events to the LED characteristic.
 *
 * @param[in] p_lbs     Instance of LED Button Service to which the write applies.
 * @param[in] led_state Written/desired state of the LED.
 */
static void led_write_handler(uint16_t conn_handle, ble_lbs_t * p_lbs, uint8_t led_state)
{
    if (led_state)
    {
        bsp_board_led_on(LEDBUTTON_LED);
        NRF_LOG_INFO("Received LED ON!");
    }
    else
    {
        bsp_board_led_off(LEDBUTTON_LED);
        NRF_LOG_INFO("Received LED OFF!");
    }
}


/**@brief Function for initializing services that will be used by the application.
 */
static void services_init(void)
{
    ret_code_t         err_code;
    ble_lbs_init_t     init     = {0};
```

```
    nrf_ble_qwr_init_t qwr_init = {0};

    // Initialize Queued Write Module.
    qwr_init.error_handler = nrf_qwr_error_handler;

    err_code = nrf_ble_qwr_init(&m_qwr, &qwr_init);
    APP_ERROR_CHECK(err_code);

    // Initialize LBS.
    init.led_write_handler = led_write_handler;

    err_code = ble_lbs_init(&m_lbs, &init);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling the Connection Parameters Module.
 *
 * @details This function will be called for all events in the Connection Parameters Module that
 *          are passed to the application.
 *
 * @note All this function does is to disconnect. This could have been done by simply
 *       setting the disconnect_on_fail config parameter, but instead we use the event
 *       handler mechanism to demonstrate its use.
 *
 * @param[in] p_evt  Event received from the Connection Parameters Module.
 */
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    ret_code_t err_code;

    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}


/**@brief Function for handling a Connection Parameters error.
 *
 * @param[in] nrf_error  Error code containing information about what went wrong.
 */
static void conn_params_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for initializing the Connection Parameters module.
 */
static void conn_params_init(void)
{
```

```
    ret_code_t             err_code;
    ble_conn_params_init_t cp_init;

    memset(&cp_init, 0, sizeof(cp_init));

    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail             = false;
    cp_init.evt_handler                    = on_conn_params_evt;
    cp_init.error_handler                  = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for starting advertising.
 */
static void advertising_start(void)
{
    ret_code_t           err_code;

    err_code = sd_ble_gap_adv_start(m_adv_handle, APP_BLE_CONN_CFG_TAG);
    APP_ERROR_CHECK(err_code);

    bsp_board_led_on(ADVERTISING_LED);
}


/**@brief Function for handling BLE events.
 *
 * @param[in]   p_ble_evt   Bluetooth stack event.
 * @param[in]   p_context   Unused.
 */
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code;

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_CONNECTED:
            NRF_LOG_INFO("Connected");
            bsp_board_led_on(CONNECTED_LED);
            bsp_board_led_off(ADVERTISING_LED);
            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
            err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr, m_conn_handle);
            APP_ERROR_CHECK(err_code);
            err_code = app_button_enable();
            APP_ERROR_CHECK(err_code);
            break;
```

```
case BLE_GAP_EVT_DISCONNECTED:
  NRF_LOG_INFO("Disconnected");
  bsp_board_led_off(CONNECTED_LED);
  m_conn_handle = BLE_CONN_HANDLE_INVALID;
  err_code = app_button_disable();
  APP_ERROR_CHECK(err_code);
  advertising_start();
  break;

case BLE_GAP_EVT_SEC_PARAMS_REQUEST:
  // Pairing not supported
  err_code = sd_ble_gap_sec_params_reply(m_conn_handle,
                    BLE_GAP_SEC_STATUS_PAIRING_NOT_SUPP,
                    NULL,
                    NULL);
  APP_ERROR_CHECK(err_code);
  break;

case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
{
  NRF_LOG_DEBUG("PHY update request.");
  ble_gap_phys_t const phys =
  {
    .rx_phys = BLE_GAP_PHY_AUTO,
    .tx_phys = BLE_GAP_PHY_AUTO,
  };
  err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
  APP_ERROR_CHECK(err_code);
} break;

case BLE_GATTS_EVT_SYS_ATTR_MISSING:
  // No system attributes have been stored.
  err_code = sd_ble_gatts_sys_attr_set(m_conn_handle, NULL, 0, 0);
  APP_ERROR_CHECK(err_code);
  break;

case BLE_GATTC_EVT_TIMEOUT:
  // Disconnect on GATT Client timeout event.
  NRF_LOG_DEBUG("GATT Client Timeout.");
  err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                 BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
  APP_ERROR_CHECK(err_code);
  break;

case BLE_GATTS_EVT_TIMEOUT:
  // Disconnect on GATT Server timeout event.
  NRF_LOG_DEBUG("GATT Server Timeout.");
  err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                 BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
  APP_ERROR_CHECK(err_code);
  break;
```

```
      default:
         // No implementation needed.
         break;
   }
}


/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
   ret_code_t err_code;

   err_code = nrf_sdh_enable_request();
   APP_ERROR_CHECK(err_code);

   // Configure the BLE stack using the default settings.
   // Fetch the start address of the application RAM.
   uint32_t ram_start = 0;
   err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
   APP_ERROR_CHECK(err_code);

   // Enable BLE stack.
   err_code = nrf_sdh_ble_enable(&ram_start);
   APP_ERROR_CHECK(err_code);

   // Register a handler for BLE events.
   NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}


/**@brief Function for handling events from the button handler module.
 *
 * @param[in] pin_no        The pin that the event applies to.
 * @param[in] button_action The button action (press/release).
 */
static void button_event_handler(uint8_t pin_no, uint8_t button_action)
{
   ret_code_t err_code;

   switch (pin_no)
   {
      case 11:
         NRF_LOG_INFO("Send button state change.");
         //check zero bit of data
         //data = (button_action & 0x1);
         //NRF_LOG_INFO("data = 0x%x",data);
         if (button_action == 1)
         {
            err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x1);
            if (err_code != NRF_SUCCESS &&
```

```
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }
        else
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x0);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }

        break;
      case 12:
       NRF_LOG_INFO("Send button state change.");
        //check zero bit of data
        //data = (button_action & 0x1);
        //NRF_LOG_INFO("data = 0x%x",data);
        if (button_action == 0x1)
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x2);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }
        else
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x0);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }

        break;
      case 24:
        NRF_LOG_INFO("Send button state change.");
        //check zero bit of data
        //data = (button_action & 0x1);
```

```
        //NRF_LOG_INFO("data = 0x%x",data);
        if (button_action == 0x1)
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x4);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }
        else
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x0);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }

        break;
      case 25:
        NRF_LOG_INFO("Send button state change.");
        //check zero bit of data
        //data = (button_action & 0x1);
        //NRF_LOG_INFO("data = 0x%x",data);
        if (button_action == 0x1)
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x8);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }
        else
        {
          err_code = ble_lbs_on_button_change(m_conn_handle, &m_lbs, 0x0);
          if (err_code != NRF_SUCCESS &&
            err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
            err_code != NRF_ERROR_INVALID_STATE &&
            err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          {
            APP_ERROR_CHECK(err_code);
          }
        }
```

```
            break;
        default:
            APP_ERROR_HANDLER(pin_no);
            break;
    }
}


/**@brief Function for initializing the button handler module.
 */
static void buttons_init(void)
{
    ret_code_t err_code;

    //The array must be static because a pointer to it will be saved in the button handler module.
    static app_button_cfg_t buttons[] =
    {
        {11, false, BUTTON_PULL, button_event_handler},
        {12, false, BUTTON_PULL, button_event_handler},
        {24, false, BUTTON_PULL, button_event_handler},
        {25, false, BUTTON_PULL, button_event_handler}

    };

    err_code = app_button_init(buttons, ARRAY_SIZE(buttons),
                               BUTTON_DETECTION_DELAY);
    APP_ERROR_CHECK(err_code);
}


static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}


/**@brief Function for initializing power management.
 */
static void power_management_init(void)
{
    ret_code_t err_code;
    err_code = nrf_pwr_mgmt_init();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling the idle state (main loop).
 *
 * @details If there is no pending log operation, then sleep until next the next event occurs.
 */
```

```
static void idle_state_handle(void)
{
   if (NRF_LOG_PROCESS() == false)
   {
      nrf_pwr_mgmt_run();
   }
}


/**@brief Function for application main entry.
 */
int main(void)
{
   // Initialize.
   log_init();
   leds_init();
   timers_init();
   buttons_init();
   power_management_init();
   ble_stack_init();
   gap_params_init();
   gatt_init();
   services_init();
   advertising_init();
   conn_params_init();

   // Start execution.
   NRF_LOG_INFO("Blinky example started.");
   advertising_start();

   // Enter main loop.
   for (;;)
   {
      idle_state_handle();
   }
}
```

Appendix E: MatLab Code for Collecting and Processing Data

```
%%MatLab Code to Get Data from Headset%%

set_param('matthewsWinner_presentation','SimulationCommand','start') %start simulation
pause(2) %wait two seconds
count=0;
while(1)
set_param('matthewsWinner_presentation','SimulationCommand','pause') %pause simulation
[N,M]=size(simout.data); %how much data collected from the simulation
for(i=1:250:N-250)
   averagedata1=sum(simout.data(i:i+249));
   averagedata=averagedata1/250; %find average over 250 data points
   if averagedata > 1000  %if average is above threshold
     writeDigitalPin(a,'D12',0) %turn on
   else
      writeDigitalPin(a,'D12',1) %turn off
   end

end
count=count+1;
set_param('matthewsWinner_presentation','SimulationCommand','continue'); %continue simulation
pause(2); % wait two seconds
end
```

Appendix F: Main Code for UART Direction Control.

```
#include <__cross_studio_io.h>
#include "nrf.h"
#include "nrf51.h"
#include "../NRF_ATM/GPIO_ATM/GPIO_ATM.h"
#include "../NRF_ATM/ADC_ATM/ADC_ATM.h"
#include "../NRF_ATM/UART_ATM/UART_ATM.h"
#include "../NRF_ATM/s140_nrf52_7.2.0_API/include/ble_gap.h"

void UART0_IRQHandler(void);

uint32_t data;

void main(void)
{
 //NRF_TIMER0->MODE&=~(0x1<<0);
 //NRF_TIMER0->MODE|=(0x0<<0); //timer mode
 //NRF_TIMER0->BITMODE&=~(0x2<<0);
 //NRF_TIMER0->BITMODE|=(0x1<<0); //set to 8 bit mode
 //NRF_TIMER0->PRESCALER&=~(0xF<<0);
 //NRF_TIMER0->PRESCALER|=(0<<0); //16 MHz clock
 //NRF_TIMER0->CC[2]=160000; //.1s
 //NRF_TIMER0->SHORTS&=~(0xF<<0)|(0xF<<8);
 //NRF_TIMER0->SHORTS|=(0x1<<2); //Clear task on cc[2] event


 uint32_t BM, PS, CCV;

 TIMER_ATM(.5, &BM, &PS, &CCV, 0);

 TIMER_CNF_ATM(0,BM,PS,CCV);

 UART_CNF_ATM(0,11,11,9600,0,0); //RXD
 UART_CNF_ATM(1,9,9,9600,0,0); //TXD

//Background PWM Signal - 1kHz
//Left Motor Driver
 NRF_TIMER1->MODE&=~(0x1<<0);
 NRF_TIMER1->MODE|=(0x0<<0); //timer mode
 NRF_TIMER1->BITMODE&=~(0x3<<0);
 NRF_TIMER1->BITMODE|=(0x0<<0); //set to 16 bit mode
 NRF_TIMER1->PRESCALER&=~(0xF<<0);
 NRF_TIMER1->PRESCALER|=(1<<0); //8 MHz clock
 NRF_TIMER1->CC[0]=6000; //.00125s
 NRF_TIMER1->CC[1]=8000; // .0025s
 //NRF_TIMER1->CC[2]=7960;
 NRF_TIMER1->SHORTS&=~(0xF<<0)|(0xF<<8);
 NRF_TIMER1->SHORTS|=(0x1<<1); //Clear task on cc[1] event

//Background PWM Signal - 1kHz
//Right Motor Driver
```

```
  NRF_TIMER2->MODE&=~(0x1<<0);
  NRF_TIMER2->MODE|=(0x0<<0); //timer mode
  NRF_TIMER2->BITMODE&=~(0x3<<0);
  NRF_TIMER2->BITMODE|=(0x0<<0); //set to 16 bit mode
  NRF_TIMER2->PRESCALER&=~(0xF<<0);
  NRF_TIMER2->PRESCALER|=(1<<0); //8 MHz clock
  NRF_TIMER2->CC[0]=6000; //.00125s
  NRF_TIMER2->CC[1]=8000; // .0025s
  //NRF_TIMER1->CC[2]=7960;
  NRF_TIMER2->SHORTS&=~(0xF<<0)|(0xF<<8);
  NRF_TIMER2->SHORTS|=(0x1<<1); //Clear task on cc[1] event

//Configure GPIO outputs for DIR of motor drivers
  GPIO_CNF_ATM(13,1,1,0,0,0); //right motor driver
  GPIO_CNF_ATM(15,1,1,0,0,0); //left motor driver

  //GPIO_WRITE_ATM(15,0); //left motor DIR //0 Forward
  //GPIO_WRITE_ATM(13,1); //right motor DIR //1 Reverse

//Configure GPIO outputs for PWN for motor driver
  GPIO_CNF_ATM(16,1,1,0,0,0); //left motor driver PWM
  GPIO_CNF_ATM(14,1,1,0,0,0); //right motor driver PWM

//Configure GPIO output for Emergency Brake relay
  GPIO_CNF_ATM(12,1,1,0,0,0);

  GPIO_WRITE_ATM(12,0);

//GPIOTE peripheral for output left motor driver
  NRF_GPIOTE->CONFIG[0]&=~((0x3<<0)|(0x1F<<8)|(0x3<<16)|(0x1<<20));
  NRF_GPIOTE->CONFIG[0]|=((0x3<<0)|(16<<8)|(0x3<<16)|(0x0<<20));
//GPIOTE peripheral for output right motor driver
  NRF_GPIOTE->CONFIG[1]&=~((0x3<<0)|(0x1F<<8)|(0x3<<16)|(0x1<<20));
  NRF_GPIOTE->CONFIG[1]|=((0x3<<0)|(14<<8)|(0x3<<16)|(0x0<<20));

//PPI peripheral for interconnecting TIMER Event to GPIO Task
  //Enable our channels for left motor driver
  NRF_PPI->CHENSET=0x1<<0;
  NRF_PPI->CH[0].EEP=&NRF_TIMER1->EVENTS_COMPARE[0];
  NRF_PPI->CH[0].TEP=&NRF_GPIOTE->TASKS_OUT[0];

  NRF_PPI->CHENSET=0x1<<1;
  NRF_PPI->CH[1].EEP=&NRF_TIMER1->EVENTS_COMPARE[1];
  NRF_PPI->CH[1].TEP=&NRF_GPIOTE->TASKS_OUT[0];

    //Enable our channels for right motor driver
  NRF_PPI->CHENSET=0x1<<2;
  NRF_PPI->CH[2].EEP=&NRF_TIMER2->EVENTS_COMPARE[0];
  NRF_PPI->CH[2].TEP=&NRF_GPIOTE->TASKS_OUT[1];

  NRF_PPI->CHENSET=0x1<<3;
  NRF_PPI->CH[3].EEP=&NRF_TIMER2->EVENTS_COMPARE[1];
  NRF_PPI->CH[3].TEP=&NRF_GPIOTE->TASKS_OUT[1];
```

```
 //NRF_TIMER1->TASKS_START=1; //Start timer 1
 //NRF_TIMER2->TASKS_START=1; //Start timer 2
 NRF_UART0->TASKS_STARTRX=1; //Start the UART
 NRF_UART0->TASKS_STARTTX=1;
 //NRF_UART0->INTENSET=1;

 NRF_UART0->INTENSET&=~(0xFF<<0);
 NRF_UART0->INTENSET|=(0x1<<2);

 NVIC_EnableIRQ(UART0_IRQn);

 //NRF_UART0->EVENTS_RXDRDY=0;
 //uint32_t data=0;
 //data=NRF_UART0->RXD;

 //61 - a
 //73 - s
 //64 - d
 //77 - w

 sd

 while(1)
 {
 //NRF_TIMER1->TASKS_STOP=1;
 //NRF_TIMER2->TASKS_STOP=1;
 __WFI();
 }



}

void UART0_IRQHandler(void)
{
NVIC_DisableIRQ(UART0_IRQn);
NRF_UART0->EVENTS_RXDRDY=0;
data=NRF_UART0->RXD&0xFF;
NRF_TIMER0->TASKS_STOP=1;
NRF_TIMER0->TASKS_CLEAR=1;

if (data==0x41) //Last byte of info for UP - 41
{
 //Start timer 0
 NRF_TIMER1->TASKS_START=1; //Start timer 1
 NRF_TIMER2->TASKS_START=1; //Start timer 2
 GPIO_WRITE_ATM(15,0); //left motor DIR //0 Forward
 GPIO_WRITE_ATM(13,0); //right motor DIR
 GPIO_WRITE_ATM(16,1);
 GPIO_WRITE_ATM(16,0);
 //NRF_TIMER0->TASKS_START=1;
//debug_printf("UP \n",data);
```

```
}
if (data==0x42) //Last byte of info for DOWN
{
  NRF_TIMER1->TASKS_START=1; //Start timer 1 Left Motor
  NRF_TIMER2->TASKS_START=1; //Start timer 2 Right Motor
  GPIO_WRITE_ATM(15,1); //left motor DIR Reverse
  GPIO_WRITE_ATM(13,1); //right motor DIR Reverse
  //NRF_TIMER1->TASKS_STOP=1;
  //NRF_TIMER2->TASKS_STOP=1;
//debug_printf("DOWN \n",data);
}
if (data==0x44) //Last byte of info for LEFT
{
  NRF_TIMER2->TASKS_START=1; //Start timer 2 Right Motor
  GPIO_WRITE_ATM(13,0); //right motor DIR forward
  //NRF_TIMER2->TASKS_STOP=1;
//debug_printf("Left \n",data);
}
if (data==0x43) //Last byte of info for RIGHT
{
  NRF_TIMER1->TASKS_START=1; //Start timer 1 Left Motor
  GPIO_WRITE_ATM(15,0); //left motor DIR Forward
  //NRF_TIMER1->TASKS_STOP=1;
//debug_printf("Right \n",data);
}

GPIO_WRITE_ATM(21,1);
NVIC_EnableIRQ(UART0_IRQn);
}
```